Universidade Federal do Espírito Santo
Departamento de Informática

Impacto de Estratégias Combinatórias no Precondicionador Paralelo baseado no Algoritmo Híbrido SPIKE

Brenno Albino Lugon

Orientadora: Profª. Drª. Lucia Catabriga
Coorientadora: Profª. Drª. Maria Cristina Rangel

22 de dezembro de 2015

Brenno Albino Lugon

Impacto de Estratégias Combinatórias no Precondicionador Paralelo baseado no Algoritmo Híbrido SPIKE

Dissertação de Mestrado

Prof[a]. Dr[a]. Lucia Catabriga
Universidade Federal do Espírito Santo
Orientadora

Prof[a]. Dr[a]. Maria Cristina Rangel
Universidade Federal do Espírito Santo
Coorientadora

Prof[a]. Dr[a]. Maria Claudia Silva Boeres
Universidade Federal do Espírito Santo

Prof. Dr. Álvaro Luiz Gayoso de Azeredo Coutinho
Universidade Federal do Rio de Janeiro

22 de dezembro de 2015

Life's a show
And we all play our parts
And when the music starts
We open up our hearts

It's all right
If some things come out wrong
We'll sing a happy song
And you can sing along

Where's there's life
There's hope, everyday's a gift
Wishes can come true
Whistle while you work
So hard all day
To be like other girls
To fit in in this glittering world

Don't give me songs
Don't give me songs
Give me something to sing about
I need something to sing about

Buffy Summers
S06E07: "Once More, With Feeling"
Buffy the Vampire Slayer

Resumo

Neste trabalho, utilizamos o algoritmo paralelo híbrido SPIKE como um precondicionador para um método iterativo não-estacionário combinando as arquiteturas de memória distribuída e compartilhada, MPI e OpenMP. A fim de obter um bom precondicionador, aplicamos um conjunto de estratégias modeladas como problemas combinatórios tais como *matching*, reordenamentos para minimizar a largura de banda da matriz, particionamento de grafos e problema quadrático da mochila que efetuam permutações nas linhas e colunas da matriz. Os experimentos computacionais demonstram a eficiência do precondicionador paralelo e a grande influência de cada uma das estratégias combinatórias apresentadas. Além disso, os resultados apresentados demonstram uma excelente escalabilidade para até 64 *cores* em aplicações de elementos finitos.

Palavras-chaves: precondicionador paralelo, SPIKE, estratégias combinatórias

Abstract

In this work, we use the parallel hybrid SPIKE algorithm as a preconditioner for a nonstationary iterative method combining distributed and shared memory architectures, MPI and OpenMP. In order to obtain a good preconditioner we apply a set of strategies modeled as combinatorial problems such as matching, reorderings to reduce matrix bandwidth, graph partitioning and the quadratic knapsack problem that perform permutations in the rows and columns of the matrix. Computational experiments demonstrate the efficiency of the parallel preconditioner and a great influence of each presented combinatorial strategies. In addition, the results show excellent scalability for up to 64 cores in finite element applications.

Keywords: parallel preconditioner, SPIKE, combinatorial strategies

Agradecimentos

Agradeço primeiramente a Deus e, em seguida, as minhas professoras orientadoras, Lucia Catabriga, Maria Cristina Rangel e Maria Claudia Boeres por todo o apoio e direcionamento que recebi durante toda minha graduação e mestrado.

Agradeço enormemente os meus pais Sergio Lugon e Gezella Albino pelo suporte da minha vida, e aos meus irmãos, Ryann Lugon, Boby Luiz, Megg Fantástico e Fred, sendo que esses três últimos se dizem cachorros.

Ao restante da minha família, minhas avós Janete e Gaida e minha tia e segunda mãe, Elza. A minha comadre Kenya e minha afilhada Manuela que constantemente me tiravam do isolamento da frente do computador, me lembrando que eu ainda tinha uma vida.

Ao meu namorado de longa data, Robson.

A minha amiga Cinthia Meireles, pelas altas discussões filosóficas interplanetárias na madrugada. Ao Marcelo Carrion e Leonardo Muniz por, além da amizade, grande ajuda na implementação e entendimento de diversas etapas deste trabalho. Agradeço ao conhecido Laboratório da diversid...Otimização, LabOtim, e todos os assíduos frequentadores com quem estive boa parte do tempo.

Agradeço a CAPES pelo auxílio financeiro, ao Laboratório de Computação Científica (LNCC) e a Universidade Federal do Rio Grande do Sul (UFRGS) por terem me concedido acesso aos seus *clusters*.

Por fim, todo mundo que de alguma forma contribuiu para a realização deste trabalho.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Sistemas lineares representam um modelo matemático de grande importância nas mais diversas áreas do conhecimento. Áreas como mecânica de fluidos computacionais, simulação de problemas físicos e químicos, matemática financeira e sistemas eletromagnéticos são alguns exemplos. Nas mais variadas aplicações, a solução de sistemas lineares resultantes é a parte computacional que demanda o maior tempo de processamento, tendo incentivado inúmeros estudos nas últimas décadas.

Nesses estudos, a classe dos métodos diretos é comumente referenciada para resolver matrizes que representam sistemas lineares. Esses métodos são capazes de encontrar a solução exata, sujeita a erros de arredondamento, em um número finito de passos. Entretanto, algoritmos dessa classe possuem um alto custo computacional equivalente a $O(n^3)$, isto é, sendo n a dimensão da matriz, esses algoritmos realizam em torno de n^3 operações de ponto flutuante. Por esse motivo, a medida que a ordem da matriz cresce, a utilização de métodos diretos pode se tornar custosa computacionalmente. Algoritmos como Eliminação de Gauss, fatoração **LU** e fatoração de *Cholesky* são alguns exemplos de métodos diretos utilizados na resolução de um sistema linear.

Com o crescimento da complexidade dos sistemas lineares e a impossibilidade dos métodos diretos de resolvê-los em tempo razoável, a classe dos métodos iterativos surgiu como uma alternativa viável. Esses métodos partem de uma solução inicial e realizam aproximações sucessivas (iterações) até convergir para a solução exata, dentro de uma tolerância especificada. Cada iteração k realiza em torno de $O(n^2)$ operações de ponto flutuante e, se atingirmos uma boa solução em um número pequeno de iterações ($k \ll n$), então esses métodos podem ser mais eficientes do que os métodos diretos.

Aplicações de diversas áreas, frequentemente, dão origem a sistemas esparsos que possuem grande quantidade de zeros. Com o objetivo de melhorar a eficiência dos

métodos diretos e iterativos, técnicas de armazenamento otimizadas visam reduzir o uso de memória e quantidade de operações de ponto flutuante armazenando apenas os elementos não nulos. Tipos de armazenamentos como o *Skyline* e *Compressed Sparse Row* (CSR) são alguns exemplos de estruturas eficientes muito usadas para resolver, respectivamente, métodos diretos e iterativos. [Saad, 2003].

O tempo de solução de sistemas de grande porte pode ainda ser otimizado através da aplicação de reordenamentos, usados para produzir um sistema com estrutura de banda estreita, isto é, um sistema onde a matriz que o representa possui uma largura de banda bem menor em relação a sua dimensão (ver Figura 1.1). A redução da largura de banda da matriz através de um reordenamento é eficiente pois, nas etapas de fatoração da matriz, ocorre o que chamamos de preenchimento (*fill-in*), onde elementos originalmente nulos se tornam diferentes de zero. Este aumento do número de elementos não nulos da matriz é prejudicial ao tempo computacional pois provoca um aumento na quantidade de operações de ponto flutuantes. No armazenamento otimizado de matrizes esparsas, onde os dados ficam dispostos em posições contíguas na memória, o *fill-in* é extremamente indesejado, uma vez que a inclusão de novos termos nessa memória requer uma computação extra, chamada de fatoração simbólica [George and Ng, 1987].

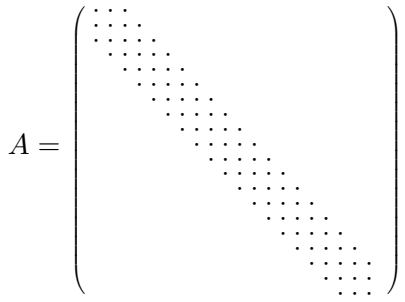


Figura 1.1: Matriz com estrutura de banda estreita

Técnicas específicas podem ser usadas para tirar vantagens de um sistema com largura de banda estreita, sendo possível criar um precondicionador robusto para um método iterativo. Precondicionadores aceleram a convergência e, consequentemente, melhoram a eficiência desses métodos na obtenção da solução [Saad, 2003]. Em aplicações paralelas, o algoritmo híbrido SPIKE, criado inicialmente para resolver um sistema linear, pode também ser usado como precondicionador para o método iterativo. Ele é dito um método híbrido por ser capaz de tirar vantagem da robustez dos métodos diretos e do baixo custo computacional dos métodos iterativos [Polizzi and Sameh, 2006].

Neste trabalho, utilizamos uma implementação paralela (MPI + OpenMP) do algoritmo híbrido SPIKE como um precondicionador do método iterativo GMRES, considerando técnicas de otimização combinatória para melhorar a qualidade desse precondicionador. As etapas que envolvem obter soluções através de métodos diretos foram computadas

utilizando o *software* PARDISO [Kuzmin et al., 2013, Schenk et al., 2007, 2008] que fornece ferramentas favoráveis ao uso de memória compartilhada, técnica conhecida como PSPIKE (PARDISO + SPIKE).

A ideia básica do algoritmo SPIKE, introduzida em 1978, é baseada na técnica de dividir e conquistar e envolvem três principais etapas: (i) pré-processamento, (ii) fatoração e (iii) pós-processamento. Neste ano, Sameh and Kuck [1978] estudaram a técnica para matrizes tridiagonais ao passo que Chen et al. [1978] avaliaram matrizes triangulares. [Lawrie and Sameh, 1984] aplicaram o algoritmo para matrizes simétricas e definidas positivas, enquanto Dongarra and Sameh [1984] consideraram o caso de matrizes estritamente diagonais dominante. Algumas variações do algoritmo SPIKE foram estudadas por Sun et al. [1992], Larriba-Pey et al. [1993] e Sun [1995]. Por fim, Polizzi and Sameh [2006, 2007] estenderam o algoritmo SPIKE para matrizes gerais com estrutura de banda e Manguoglu et al. [2009] desenvolveram o PSPIKE utilizando o *software* de memória compartilhada PARDISO.

Com o objetivo de melhorar a qualidade do precondicionador, implementamos neste trabalho um conjunto de técnicas combinatórias. As técnicas estão relacionadas a objetivos específicos, que precisam ser alcançados para o bom funcionamento do SPIKE, e são modeladas como algoritmos em grafos, gerando permutações que, quando aplicadas na matriz, transformam-na em uma matriz com propriedades convenientes. Dentre os objetivos que desejamos alcançar estão: (i) mover os elementos não nulos para a diagonal principal através de um *matching* perfeito em grafos realizado pelo algoritmo *weighted bipartite matching* [Duff and Koster, 1999], (ii) reduzir a largura de banda da matriz original através dos reordenamentos Espectral [Barnard et al., 1993] e Espectral Valorado [Manguoglu et al., 2010], (iii) particionar os dados da matriz em blocos para os processadores distribuídos utilizando o modelo *Chains-on-chains partitioning* para realizar um particionamento em grafos [Pinar and Aykanat, 2004] e, por fim, (iv) mover os elementos mais significativos para dentro dos blocos do algoritmo SPIKE, estratégia modelada como o problema quadrático da mochila e executada pelo algoritmo *DeMin* [Naumann and Schenk, 2012]. Um dos principais objetivos deste trabalho é analisar detalhadamente o comportamento de cada uma dessas técnicas combinatórias e suas influências no número de iterações e tempo de processamento do método iterativo GMRES com o precondicionador SPIKE.

Os capítulos deste trabalho estão divididos de forma a facilitar a compreensão do leitor. No segundo capítulo, revisamos alguns conceitos básicos sobre sistemas de equações lineares, matrizes esparsas, reordenamentos e teoria dos grafos. No terceiro capítulo, descrevemos sobre as arquiteturas paralelas de memória compartilhada e memória distribuída, assim como uma abordagem híbrida dessas duas arquiteturas. No capítulo

4, apresentamos todo o embasamento teórico envolvido no algoritmo SPIKE, desde sua abordagem clássica até seu uso como um precondicionador paralelo e em seguida, no quinto capítulo, mostramos o conjunto de estratégias combinatórias utilizadas. No sexto capítulo, descrevemos as técnicas e detalhes de implementação. No capítulo 7, exibimos os experimentos computacionais com uma análise de resultados e, por fim, as conclusões e trabalhos futuros.

Capítulo 2

Conceitos Básicos

Algumas definições importantes são necessárias para o bom entendimento deste trabalho. Neste capítulo apresentamos um resumo dos conceitos mais relevantes de álgebra linear, técnicas de solução de sistemas lineares esparsos utilizando precondicionadores e armazenamento otimizado, conceitos fundamentais da teoria dos grafos e transformações de matrizes através de permutações.

2.1 Sistemas Lineares

Seja uma equação linear de n incógnitas definida como $a_{i1}x_1 + a_{i2}x_2 + ... + a_{in}x_n = b_i$, um sistema linear é um conjunto de m equações lineares da forma

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n &= b_2 \\ \vdots & \\ a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n &= b_m \end{cases}$$

onde a_{ij}, $b_i \in \mathbb{R} \cup \mathbb{C}$, para $1 \leq i \leq m, 1 \leq j \leq n$, são os coeficientes do sistema.

Assim, podemos representar um sistema linear na forma matricial $Ax = b$

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}.$$

A matriz A é chamada de matriz do sistema, os vetores coluna x e b são, respectivamente, as incógnitas e os termos independentes.

Seja $A = \{a_{ij}\}$ uma matriz. A é dita *quadrada* se possui o mesmo número de linhas e colunas. A é *diagonal* se $a_{ij} = 0$, para $i \neq j$ e é dita *diagonal dominante* se $\sum_{i \neq j} |a_{ij}| \leq |a_{ii}|\ \forall i$. A *matriz transposta de* A, denotada por A^T, é tal que $a_{ji}^T = a_{ij}$. A é chamada de *matriz triangular superior (U)* se $a_{ij} = 0$ para $i > j$ e, *matriz triangular inferior (L)* se $a_{ij} = 0$ para $i < j$. A é dita *identidade* se $a_{ij} = 1$ quando $i = j$ e $a_{ij} = 0$ quando $i \neq j$. A é *simétrica* se $A = A^T$ e é *estruturalmente simétrica* se, quando $a_{ij} \neq 0$ então $a_{ji} \neq 0$, mas não necessariamente $a_{ij} = a_{ji}$. Seja um vetor $v \neq 0$, e um escalar $\lambda \in \mathbb{R}$ tais que $Av = \lambda v$, λ é um *autovalor* de A e v um *autovetor* de A associado a λ. Uma matriz A é *definida positiva* se $v^T Av > 0, \forall v \neq 0$ e $v \in \mathbb{R}^n$. Uma matriz quadrada é chamada de *identidade* I quando todos os elementos da diagonal principal são iguais a 1 e os demais elementos são nulos. A matriz *inversa* de A, denotada A^{-1}, é tal que $A.A^{-1} = I$ e $A^{-1}.A = I$. Uma matriz é dita *não-singular* se ela for invertível, ou seja, se ela admite uma inversa.

A solução de um sistema linear pode ser encontrada através de métodos diretos ou métodos iterativos. Os métodos diretos são capazes de encontrar a solução exata, sujeita a erros de arredondamento, em um número finito de passos. Os métodos iterativos são técnicas mais eficazes para resolver sistemas lineares de grande porte uma vez que comumente realizam menos operações. Sendo assim, proporcionam uma boa qualidade de solução quanto a erros de arredondamento, visto que não alteram a matriz A e o vetor b durante o processo iterativo [Saad, 2003].

Métodos iterativos não-estacionários baseados em subespaços de *Krylov*, tais como GMRES e BiCGStab, são bons candidatos para o uso do precondicionador baseado no algoritmo SPIKE. Essa classe de algoritmos procura obter, em cada iteração, a melhor aproximação utilizando informações das iterações anteriores. Tais métodos se baseiam em projeções de subespaços de *Krylov*, formados a partir de

$$K_m(A, r_0) = span\left\{r_0, Ar_0, A^2r_0, ..., A^{m-1}r_0\right\}, \tag{2.1}$$

onde m é a dimensão do subespaço, $r_0 = b - Ax_0$ e $span\left\{r_0, Ar_0, A^2r_0, ..., A^{m-1}r_0\right\}$ representa a base geradora do subespaço vetorial K_m. O objetivo é transformar o sistema $Ax = b$ em um problema de minimização do resíduo $r = b - Ax$, para $x \in K_m$.

Para os experimentos realizados neste trabalho, optamos pelo uso do método iterativo não-estacionário GMRES (Método do Resíduo Mínimo Generalizado) desenvolvido por Saad and Schultz [1986]. O algoritmo utiliza o processo de Arnoldi para calcular uma base ortonormal do subespaço de *Krylov*. A base ortonormal x_k é dada por $x_0 + Vy_k$, onde

as colunas da matriz V são os vetores v_k calculados pelo processo de Arnoldi. O vetor y_k é obtido pela solução do sistema $H_k y_k = \beta_k e_1$, sendo H_k uma matriz triangular superior de *Hessenberg* calculada durante a ortogonalização do processo de Arnoldi, $\beta_k = \|r_0\|_2$, $r_0 = b - Ax_0$ e e_1, o vetor canônico de dimensão k [Gonçalez, 2005].

A fim de reduzir o número de operações de ponto flutuante e o uso de memória, o método iterativo pode ser implementado considerando um número fixo de elementos na base. Este procedimento é conhecido como reinicialização (*restart*) [Saad and Schultz, 1986] e funciona gerando, a cada k iterações, um novo espaço vetorial de *Krylov* com k elementos na base. Para valores grandes de k, o método fica mais robusto e converge com menos iterações. Entretanto, se k for muito grande, o objetivo do *restart* de reduzir os custos computacionais e de armazenamento é invalidado. Valores pequenos de k são, frequentemente, boas escolhas pois, apesar do método necessitar de mais iterações, cada iteração é menos custosa. Na prática, escolher um valor apropriado para k pode ser uma tarefa árdua. É importante conhecer as características do problema e encontrar um ponto de equilíbrio entre boas taxas de convergência e redução de custos computacionais.

2.1.1 Precondicionamento

Com objetivo de acelerar a convergência dos métodos iterativos baseados em espaços de *Krylov*, o precondicionamento consiste em encontrar uma matriz M que seja simples de construir e que proporcione a redução do número de iterações de um sistema linear. Um bom precondicionador M deve ser uma aproximação de A de forma que, resolver o sistema linear com precondicionador à esquerda

$$M^{-1}Ax = M^{-1}b \tag{2.2}$$

ou à direita

$$AM^{-1}u = b, \text{ com } x = M^{-1}u \tag{2.3}$$

melhore a convergência do método iterativo.

Para o precondicionamento do método GMRES, consideramos neste trabalho o precondicionador à esquerda. Algebricamente, aplicando o precondicionador no sistema $Ax = b$, temos o sistema equivalente da Equação 2.2. No processo iterativo, a influência do precondicionador acontece em cada produto matriz-vetor, ou seja, cada operação do tipo $p = Ax_i$ é substituída pela operação $p = M^{-1}Ax_i$.

Entretanto, a fim de evitar o cálculo da matriz inversa de M, a operação $p = M^{-1}Ax_i$ não é computada explicitamente. Na prática, esta operação é feita em duas etapas:

1. calcula-se $v = Ax_i$
2. calcula-se $p = M^{-1}v$, resolvendo o sistema $Mp = v$.

O algoritmo híbrido SPIKE, criado inicialmente para resolver um sistema linear, pode ser usado como um método direto ou como um precondicionador para métodos iterativos. É considerado um método híbrido por ser capaz de tirar vantagem da robustez dos métodos diretos e do baixo custo computacional alcançado pelos métodos iterativos. De acordo com Polizzi and Sameh [2006], algumas de suas versões são mais rápidas do que métodos diretos presentes na ScaLapack, uma biblioteca de alto desempenho que implementa rotinas de álgebra linear para máquinas paralelas com memória distribuída.

O SPIKE clássico, detalhado na seção 4.1, se baseia na decomposição $A = D \times S$. No caso do precondicionador SPIKE, as matrizes D e S não são calculadas de forma exata, visto que muitos elementos podem ser desconsiderados. Logo, podemos definir a decomposição da matriz precondicionadora do SPIKE como

$$M = \tilde{D} \times \tilde{S} + R \,, \tag{2.4}$$

onde R é a matriz de resíduo que representa os elementos descartados do precondicionador. Dependendo do quão próximo do exato forem as matriz $\tilde{D}$ e $\tilde{S}$, mais influência o precondicionador terá na resolução do sistema. Se R for zero, então a solução do sistema é imediata, caso contrário, é necessário algumas correções na solução. Essas correções serão computadas pelo método iterativo Gmres.

2.2 Matrizes Esparsas

Saad [2003] considera que uma matriz é esparsa quando podemos tirar vantagens da grande quantidade de zeros, uma vez que estes não precisam ser armazenados. Para isso, é preciso definir estruturas de dados adequadas para uma implementação eficiente de métodos numéricos de solução, sejam eles diretos ou iterativos.

2.2.1 Armazenamento Otimizado

Uma forma de reduzir os gastos computacionais de operações em matrizes esparsas é otimizar o armazenamento dessa matriz. Neste trabalho utilizamos o método chamado *Compress Sparse Row* (CSR). Através desta técnica, o armazenamento completo da matriz A é substituído por três vetores auxiliares: AA, JA e IA. Os dois primeiros vetores, AA e JA, armazenam as contribuições não nulas, linha a linha, e a coluna

correspondente que cada uma ocupa em A, respectivamente. O vetor IA informa a posição em AA do primeiro elemento não nulo de cada linha de A com seu último elemento sendo igual ao número de elementos não nulos acrescido de um.

A Figura 2.1 mostra um exemplo do armazenamento CSR. A matriz A escrita na sua forma completa (esquerda) é armazenada nos vetores AA, JA e IA (direita).

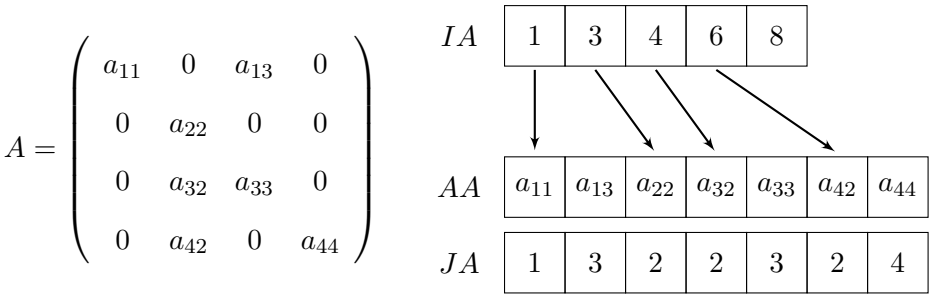


Figura 2.1: Armazenamento CSR

2.2.2 Largura de Banda

Uma métrica muito importante para entendermos os algoritmos descritos neste trabalho é chamada de largura de banda. Seja A uma matriz estruturalmente simétrica, a *largura de banda* de A é definida como

$$lb(A) = \max_{i=1,..,n} \{b_i\} \tag{2.5}$$

$$b_i = (i-j) \ \forall \ a_{ij} \neq 0, \ i = 1, ..., n \ . \tag{2.6}$$

Em outras palavras, podemos dizer que a largura de banda é a maior distância entre o primeiro elemento não-nulo da linha i até a diagonal principal. Na Figura 2.2 podemos ver um exemplo desta métrica em uma matriz.

	1	2	3	4	5	6	7	8	9	10
1	×		×					×		
2		×			×					
3	×		×				×			
4				×					×	×
5		×			×				×	
6						×	×	×		
7			×			×	×			
8	×					×		×	×	×
9				×	×			×	×	
10				×				×		×

Figura 2.2: Largura de Banda = 7

2.2.3 *Fill-in*

O processo de fatoração **LU** muito utilizado em métodos diretos de solução de sistemas lineares, quando aplicado em uma matriz esparsa, eventualmente tende a preencher muitos elementos nulos, diminuindo consideravelmente o grau de esparsidade da matriz. Com isso, o tempo computacional para resolver sistemas que utilizam essa matriz aumenta, uma vez que cresce o número de operações de ponto flutuante.

Muitas heurísticas de reordenamento de matrizes foram criadas para reduzir o preenchimento (ou *fill-in*) de matrizes esparsas. Algoritmos como o *Nested Dissection* ou de Grau Mínimo visam simplesmente reduzir o *fill-in*, enquanto algoritmos como o *Reverse Cuthill McKee* (RCM), *Sloan* e Espectral tentam aproximar os elementos não nulos da diagonal principal o que, consequentemente, reduzem o *fill-in*.

Na Figura 2.3 podemos observar como o *fill-in* acontece. A fatoração **LU** aplicada na matriz com a configuração de (a) gera muito *fill-in* em (b), onde cada ■ representa um novo elemento não nulo. Em (c), a simples troca das linhas e colunas 1 e 5 de (a) reduz o *fill-in* a zero na matriz representada em (d).

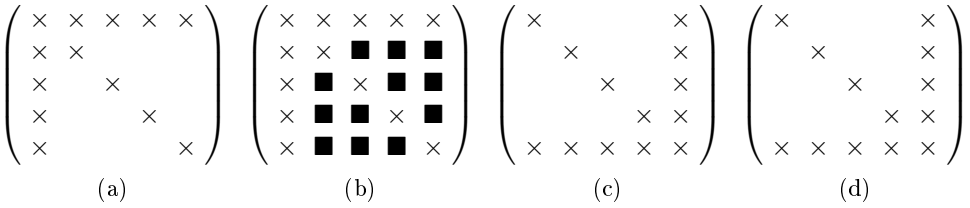


Figura 2.3: Exemplo de como ocorre o *fill-in*

2.3 Grafos

Conceitos importantes em teoria de grafos são fundamentais para o entendimento deste trabalho. Isto porque as estratégias combinatórias apresentadas no Capítulo 5 são modeladas como algoritmos em grafos.

Seja $G = (V, E)$ um grafo formado pelos vértices $V = \{v_1, v_2, v_3, ..., v_n\}$ e um conjunto de arestas $E = \{e_1, e_2, e_3, ..., e_m\}$, onde $e_i = v_i v_j$, dizemos que um vértice é *adjacente* a outro se há uma aresta que é *incidente* nesses dois vértices. O conjunto de *vizinhos* de um vértice consiste de todos os vértices adjacentes a ele. O número de vizinhos de $v \in V$ é o *grau* do vértice v, denotado por $d_G(v)$.

Considere o subconjunto de vértices $X = x_1, x_2, ..., x_n \subseteq V$ e suponha que a aresta $x_i x_{i+1} \in E$ para todo $i \in 1, 2, ..., k-1$. Se os vértices de X são todos distintos, diz-se

que X define um *caminho*. Um *ciclo elementar* é um caminho com mesmo vértice final e inicial. Um grafo é dito *conexo* se possui um caminho entre qualquer par de vértices. Uma *árvore* é um grafo conexo sem ciclos. Definimos *estrutura de nível* $EN(v) = N_1, N_2, ..., N_k$ uma árvore com níveis denotados por $N_1, N_2, ..., N_k$.

Seja $v, w \in V(G)$, denomina-se *distância* $d(v, w)$ de um grafo como sendo o comprimento do menor caminho entre v e w. A *excentricidade* de um vértice v é a maior distância de v a todos os outros vértices de G. Definimos *diâmetro* de G como sendo a maior excentricidade do grafo G e o *pseudo-diâmetro* é uma excentricidade alta de G, porém não necessariamente a maior. Dado um grafo G, os *vértices periféricos* de G são vértices cuja excentricidade é igual ao diâmetro de G. Vértices com altas excentricidades, porém não necessariamente a maior, definem os *vértices pseudo-periféricos*.

Dados dois grafos $F = (V_F, E_F)$ e $G = (V_G, E_G)$, F é um *subgrafo* de G quando $V_F \subseteq V_G$ e $E_F \subseteq E_G$. Um grafo $G = (V, E)$ diz-se *bipartido* se seu conjunto de vértices admite uma partição em subconjuntos V_1 e V_2 tal que não existem arestas que unem dois vértices da partição V_1 ou dois vértices da partição V_2.

Um hipergrafo H é uma generalização de um grafo e pode ser definido como sendo um par ordenado $H = (V, E)$ onde V é um conjunto de vértices e E é um conjunto de arestas onde E é formado por conjuntos não vazios de V. Em outra palavras, uma aresta nada mais é do que um conjunto de vértices. A Figura 2.4 mostra a representação de um hipergrafo onde o conjunto de vértices e arestas são $V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$ e $E = \{e_1, e_2, e_3, e_4\} = \{\{v_1, v_2, v_3\}, \{v_2, v_3\}, \{v_3, v_5, v_6\}, \{v_4\}\}$, respectivamente.

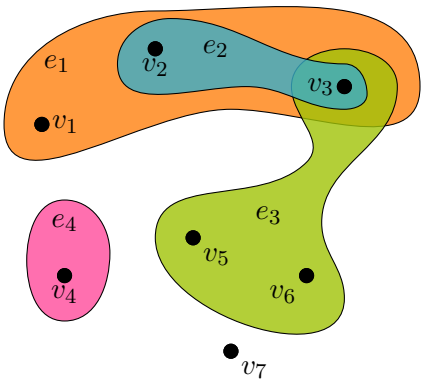


Figura 2.4: Representação de um hipergrafo

Definimos *matriz de adjacência* de um grafo $G = (V, E)$, a matriz $A_G = a_{ij}$, tal que

$$a_{ij} = \begin{cases} 1 & \text{se } (i, j) \in E \\ 0 & \text{caso contrário} \end{cases}$$

Por fim, a matriz *Laplaciana* L de um grafo $G = (V, E)$ é definida como $L = D - A$, D é uma matriz diagonal contendo os graus de cada vértice v_i do grafo G e A é a matriz de adjacência de G. Ou seja,

$$L_{ij} = \begin{cases} grau(v_i) & \text{se } i = j \\ -1 & \text{se } i \neq j \text{ e } v_i \text{ é adjacente a } v_j \\ 0 & \text{caso contrário} \end{cases}$$

Os conceitos de grafos descritos acima são muito importantes pois formam a base para as estratégias combinatórias implementadas neste trabalho, no momento em que podemos representar uma matriz como um grafo. De maneira geral, uma matriz $n \times n$ pode ser representada por um grafo de n vértices, onde existe a aresta (i, j) quando a_{ij} é diferente de zero. A Figura 2.5 mostra um exemplo de equivalência entre a matriz e o grafo.

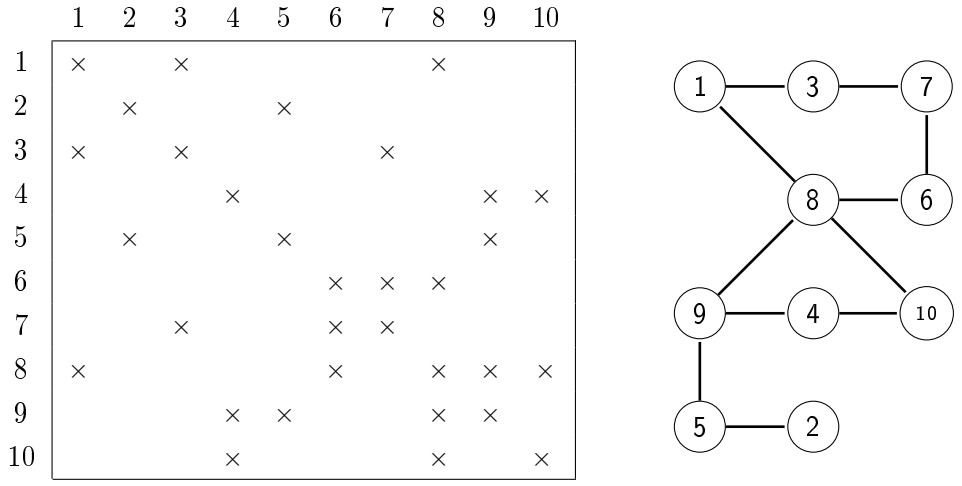


Figura 2.5: Matriz e grafo correspondente

2.4 Permutações de Matrizes

Permutações de linhas ou colunas são operações muito utilizadas na manipulação de matrizes. Sendo P uma matriz identidade e $\boldsymbol{\pi}$ um vetor de permutação, definimos a matriz de permutação aplicando o vetor $\boldsymbol{\pi}$ em P. Por exemplo, considere o vetor de permutação $\boldsymbol{\pi} =$ (4 2 3 1 5). Ao aplicarmos esse vetor na matriz identidade, temos a matriz de permutação P como sendo

$$P = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} .$$

As estratégias combinatórias utilizadas neste trabalho visam aplicar permutações na matriz original de forma que a matriz permutada apresente melhores propriedades matemáticas e computacionais. Dois tipos de permutações podem ser aplicadas:

1. Simétrica
2. Não simétrica

A permutação simétrica consiste em aplicar a matriz de permutação P trocando, simultaneamente, linhas e colunas de A. Uma permutação simétrica preserva a simetria da matriz. Além disso, todos os elementos da diagonal principal permanecem na diagonal principal, em ordem distintas. O sistema linear resultante após a aplicação de uma permutação simétrica é

$$PAP^T\ Px = Pb\ . \tag{2.7}$$

A permutação não simétrica aplica a matriz de permutação P somente nas linhas ou somente nas colunas de A. Uma matriz simétrica submetida a essa permutação perde tal propriedade, entretanto, mesmo perdendo a propriedade de simetria, a permutação pode ser uma ação positiva ao viabilizar uma estratégia de solução mais vantajosa. Este fato justifica o uso do método iterativo GMRES, ao contrário de um método mais adequado como o Gradiente Conjugado (GC) usado para encontrar a solução de sistemas lineares de matrizes simétricas e definidas positivas. Diferente da permutação simétrica, a permutação não simétrica retira os elementos presentes originalmente na diagonal principal. O sistemas lineares resultantes desta permutação são

$$PA\ x = Pb \tag{2.8}$$

para uma permutação somente de linhas e

$$AP^T\ Px = b \tag{2.9}$$

para uma permutação somente de colunas.

Capítulo 3

Programação Paralela

Tradicionalmente, computadores foram desenvolvidos para executar instruções serialmente na Unidade Central de Processamento ou CPU (*Central Processing Unit*). Nessa abordagem, apenas uma instrução é executada a cada passo de tempo — após o término dessa instrução, a próxima é executada [Barney, 2007].

Motivada pelo aumento da complexidade dos problemas computacionais e da necessidade de resolvê-los de maneira mais eficiente e rápida, surgiu a programação paralela. Essencialmente, a programação paralela consiste na utilização simultânea de recursos computacionais para resolver um determinado problema. Supõe-se antecipadamente que este problema pode ser dividido em partes menores e independentes, para que cada uma dessas partes possa ser computada concorrentemente, ou seja, em paralelo.

A maneira mais usual de avaliar a eficiência de um programa paralelo é através de uma medida que chamamos de *speedup*. De acordo com Denning and Tichy [1990],

> "*speedup* é uma medida comum do ganho de desempenho de um processador paralelo. Ele é definido como a razão entre o tempo necessário para completar uma tarefa com um processador e o tempo necessário para concluir esta tarefa com N processadores."

Sendo assim,

$$speedup = \frac{T_{serial}}{T_{paralelo}} \tag{3.1}$$

onde, T_{serial} é o tempo de execução do programa serial e $T_{paralelo}$ é o tempo de execução do programa paralelo [Karp and Flatt, 1990].

Uma das classificações existentes quanto a arquitetura de programas paralelos é com relação ao uso de memória. A seção 3.1 apresenta características da biblioteca MPI, desenvolvida para sistemas com memória distribuída, a seção 3.2 expõe o paradigma OpenMP, desenvolvido para sistemas com memória compartilhada e, por fim, a seção 3.3 discute uma abordagem híbrida entre essas duas arquiteturas de memória.

3.1 *Message Passing Interface*

Message Passing Interface (MPI) é uma biblioteca desenvolvida para programação paralela usando memória distribuída. Também chamados de multicomputadores ou *clusters*, cada processador nesta arquitetura tem acesso somente a sua memória local e a comunicação com os outros processadores é realizada através de troca de mensagens, como mostra a Figura 3.1.

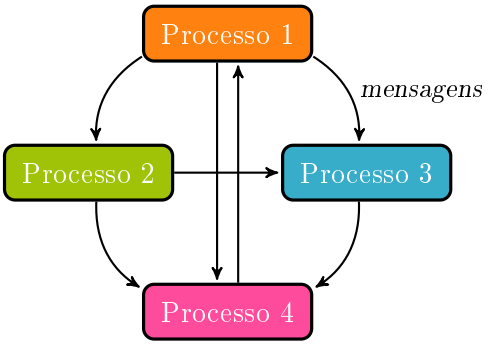


Figura 3.1: Comunicação distribuída via troca de mensagens

Na biblioteca MPI, os vários processos executam o mesmo fluxo de instruções, e cada processo é identificado por um identificador (*rank*). As grandes vantagens dessa interface estão na sua praticidade, eficiência, flexibilidade e portabilidade, com suporte para programação nas linguagens C/C++ e Fortran, em suas versões mais recentes. Dentre as desvantagens estão o tempo (*overhead*) causado na comunicação através da rede e a necessidade de reestruturação do código sequencial. A Tabela 3.1 descreve as principais rotinas disponíveis para suporte à programação MPI.

Rotina	Descrição
MPI_Init	Inicializa o ambiente MPI
MPI_Comm_size	Retorna a quantidade de processos
MPI_Comm_rank	Retorna o *rank* do processo
MPI_Send	Envia uma mensagem
MPI_Recv	Recebe uma mensagem
MPI_Finalize	Finaliza o ambiente MPI

Tabela 3.1: Principais rotinas do MPI

3.2 *Open Multi-Processing*

Open Multi-Processing (OpenMP) é uma interface de programação desenvolvida para computadores com multiprocessadores ou SMP (*Symmetric Multi-Processors*) que compartilham principalmente a memória, dentre outros recursos. O modelo de memória compartilhada é caracterizado por possuir uma memória única que pode ser acessada simultaneamente por múltiplos processadores, chamados *threads*, como mostra a Figura 3.2. A comunicação entre as *threads* é bastante rápida neste modelo de programação, pois os dados escritos na memória compartilhada podem ser lidos por qualquer *thread*, simplesmente acessando posições desta memória.

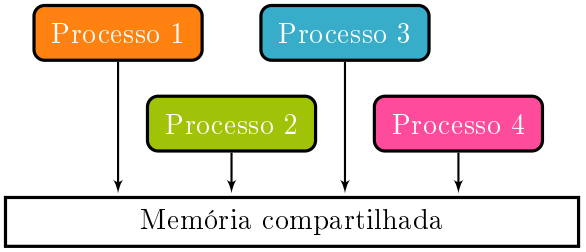


Figura 3.2: Comunicação via memória compartilhada

O OpenMP é uma aplicação *multithreading* baseada no modelo *Fork-Join*, ou seja, uma *thread* mestre subdivide uma determinada tarefa para um número específico de *threads* que serão executadas em paralelo. A Figura 3.3 esquematiza o modelo *Fork-Join*: a *thread* mestre cria um conjunto de outras *threads* (*fork*) que executarão o programa na região paralela. Após concluírem suas tarefas, as *threads* terminam sua execução e os resultados são sincronizadas (*join*), restando apenas a *thread* mestre.

Aplicações usando OpenMP podem ser bem simples de implementar, inserindo diretivas do compilação — que podem ser incorporadas em um código fonte C/C++ e Fortran — no código serial. Apesar dessa vantagem, o OpenMP não é tão escalável como o MPI e pode haver perda de desempenho dependendo da quantidade de acessos concorrentes na memória compartilhada. No contexto de programação paralela, podemos dizer que escalabilidade é a capacidade de aumentar o desempenho computacional proporcionalmente ao aumento da quantidade de processadores.

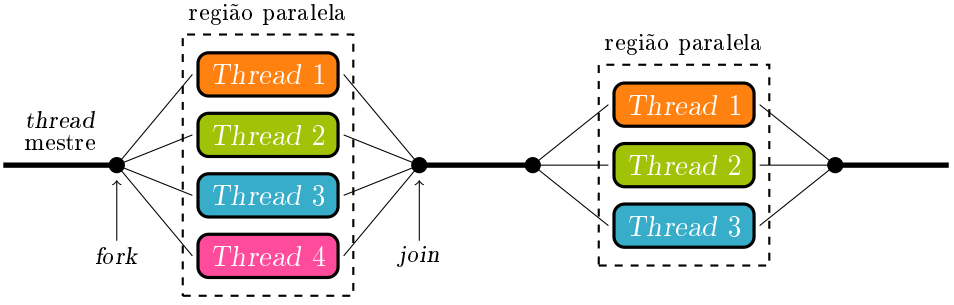


Figura 3.3: Modelo *Fork-Join*

3.3 Abordagem Híbrida MPI e OpenMP

Com o crescimento na área de programação paralela e evolução das citadas arquiteturas, uma tendência é o uso de uma abordagem híbrida entre MPI e OpenMP. Neste paradigma de programação, multicomputadores exploram o paralelismo usando memória distribuída enquanto que multiprocessadores garantem as vantagens do uso de memória compartilhada. A Figura 3.4 exibe o esquema arquitetural dessa abordagem híbrida. Diferentes nós (Memória + 4 *cores*) se comunicam usando MPI via rede, enquanto que os *cores* em cada nó se comunicam usando OpenMP.

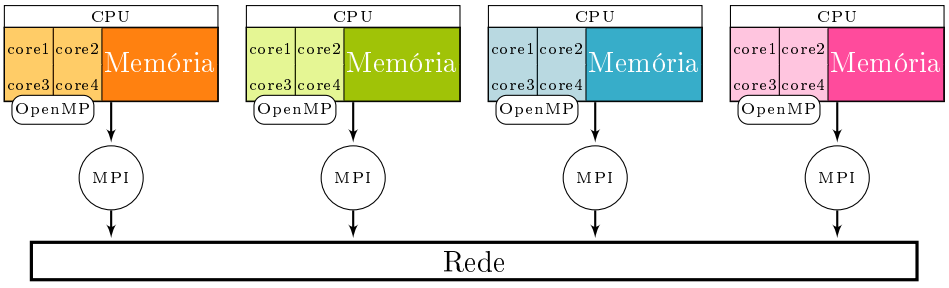


Figura 3.4: Abordagem híbrida MPI e OpenMP

A combinação de diferentes níveis de paralelismo tem sido bastante explorada em trabalhos acadêmicos recentes, por exemplo, para cálculos de escoamento incompressível [Jacobsen and Senocak, 2013] ou para alcançar uma baixa latência no reconhecimento de atividades em *streaming* de video [Chen et al., 2010]. Um conceito abstrato desta abordagem paralela multinível é a granularidade — quantidade de computação realizada em uma aplicação paralela. Podemos dizer acerca do paralelismo em multicomputadores com MPI como sendo de granularidade grossa, e com os multiprocessadores do OpenMP, de granularidade fina. Esta última é aplicada quando podemos dividir a carga de trabalho em grandes quantidades de tarefas menores. Uma granularidade muito fina seria, teoricamente, ideal na exploração do paralelismo e no aumento do *speedup*, porém, a grande quantidade de processos causaria um *overhead* indesejado durante a comunicação e sincronização desses processos. Já uma aplicação é dita de granularidade grossa quando cada processo tem uma carga maior de trabalho antes de realizar uma comunicação. Assim, a melhor estratégia para se obter a melhor performance paralela é encontrar um equilíbrio entre a quantidade de trabalho por processo e o *overhead* causado pela comunicação.

Neste trabalho, alguns experimentos utilizaram essa abordagem híbrida MPI + OpenMP em diversas configurações de processadores, com o objetivo de verificar as melhores escolhas e analisar escalabilidade e *speedup*.

Capítulo 4

Algoritmo híbrido SPIKE

4.1 Algoritmo Clássico

O algoritmo paralelo híbrido SPIKE tem como objetivo encontrar a solução de um sistema linear da forma $Ax = f$ usando computação paralela. Sendo $A = \{a_{ij}\}_{n \times n}$ uma matriz esparsa não-simétrica e diagonal dominante, podemos dividir o algoritmo SPIKE em três etapas principais: pré-processamento, fatoração e pós-processamento.

4.1.1 Pré-processamento

A primeira etapa do pré-processamento consiste em transformar uma matriz esparsa geral em uma matriz com estrutura de banda, i.e., com os elementos não nulos dispostos próximos à diagonal principal e com uma largura de banda b pequena ($b \ll n$). Transformar a matriz geral em uma matriz com estrutura de banda é uma condição necessária para que o método SPIKE possa ser aplicado. Esta meta pode ser alcançada através de um reordenamento, que será detalhado na Seção 5.2.

Após o reordenamento, a próxima etapa é particionar a matriz com estrutura de banda em uma matriz bloco tridiagonal que possui a forma

$$\begin{pmatrix} A_1 & B_1 & & & & & \\ C_2 & A_2 & B_2 & & & & \\ & \ddots & \ddots & \ddots & & & \\ & & C_k & A_k & B_k & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & C_{n-1} & A_{n-1} & B_{n-1} \\ & & & & & C_n & A_n \end{pmatrix}.$$

Esta matriz é dita como um tipo especial de matriz, formada pelas submatrizes quadradas (blocos) C_k, A_k e B_k — que podem ter dimensões distintas — dispostas na diagonal inferior, principal e superior, respectivamente, e com o restante dos elementos iguais a zero. Diferente de uma matriz tridiagonal, a matriz bloco tridiagonal é formada por submatrizes ao invés de escalares.

A Figura 4.1 mostra, conceitualmente, um particionamento, em quatro processadores, de uma matriz com estrutura de banda em uma matriz bloco tridiagonal adequada para ser utilizada pelo algoritmo SPIKE. Particionar a matriz A a fim de se obter os blocos C_i, A_i e B_i pode ser interpretado como um problema de particionamento de grafos (ver Seção 5.3, Estratégias Combinatórias). Maiores detalhes sobre a construção das matrizes C_i, A_i e B_i serão apresentados na próxima seção.

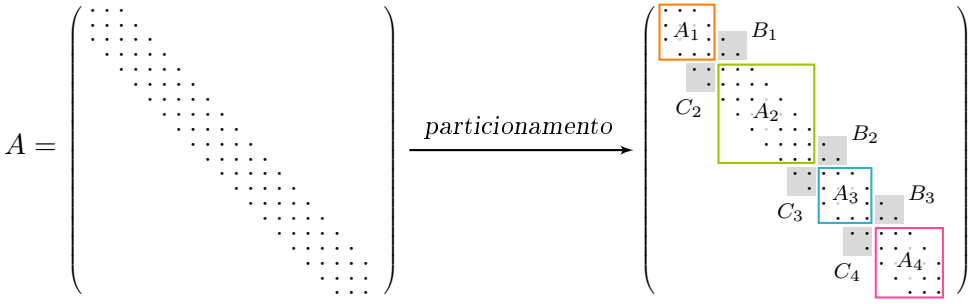


Figura 4.1: Particionamento bloco tridiagonal em 4 processadores

4.1.2 Fatoração

Diferente das fatorações mais utilizadas pelos métodos diretos, tais como $A = LU$ ou $A = LDL^T$, o algoritmo SPIKE se baseia na fatoração $A = D \times S$, onde D é uma matriz bloco diagonal e S é chamada de matriz de *spikes*. A Figura 4.2 mostra essa fatoração após um particionamento em quatro processadores.

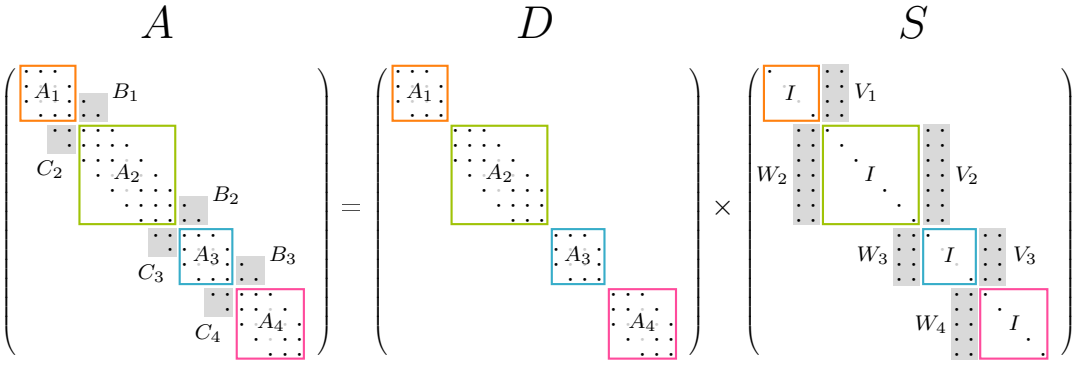


Figura 4.2: Decomposição $A = D \times S$

A matriz A é formada pelas matrizes bloco diagonais A_i $(i = 1, ..., p)$ e pelas matrizes bloco de acoplamento B_i $(i = 1, ..., p-1)$ e C_i $(i = 2, ..., p)$. Para garantir que todos os elementos não nulos dispostos fora das matrizes bloco diagonais sejam cobertos pelas matrizes bloco de acoplamento, B_i e C_i possuem dimensão $\mathbf{k} \times \mathbf{k}$, onde $\mathbf{k}$ é a largura de banda de A. A matriz D é formada pelas matrizes bloco diagonais quadradas A_i e, assumindo a não singularidade de cada bloco A_i, a matriz $S = D^{-1}A$ é formada pelas matrizes densas (*spikes*) V_i $(i = 1, ..., p-1)$ e W_i $(i = 2, ..., p)$, de dimensão $n_i \times \mathbf{k}$, sendo n_i o número de linhas dos blocos A_i.

As matrizes *spikes* V_i e W_i são dadas por

$$V_i = A_i^{-1} \begin{pmatrix} 0 \\ B_i \end{pmatrix} \text{ e } W_i = A_i^{-1} \begin{pmatrix} C_i \\ 0 \end{pmatrix}, \tag{4.1}$$

e podem ser calculadas resolvendo os sistemas

$$A_i V_i = \begin{pmatrix} 0 \\ B_i \end{pmatrix} \text{ e } A_i W_i = \begin{pmatrix} C_i \\ 0 \end{pmatrix}, \tag{4.2}$$

onde o primeiro processador calcula apena V_1 e o último processador p calcula apenas W_p. Note que os sistemas da Equação 4.2 possuem múltiplos vetores de termos independentes e, consequentemente, múltiplos vetores como solução. Esses sistemas são resolvidos utilizando o método direto de fatoração $\mathbf{LU}$, com $A_i = \mathbf{L}_i\mathbf{U}_i$, e podem ser representados na forma reduzida por

$$\mathbf{L}_i\mathbf{U}_i[V_i, W_i] = \left[\begin{pmatrix} 0 \\ B_i \end{pmatrix}, \begin{pmatrix} C_i \\ 0 \end{pmatrix} \right] . \tag{4.3}$$

4.1.3 Pós-processamento

O sistema $Ax = f$ com a fatoração $A = D \times S$ deve ser resolvido em duas etapas:

1. resolver $Dg = f$
2. resolver $Sx = g$

O sistema $Dg = f$ pode ser resolvido em paralelo sem gastos com comunicação e sincronização entre os processadores. Isso porque a matriz D é formada apenas pelas matrizes bloco diagonais A_i, independentes entre si. Portanto, uma das formas de resolver esta etapa é através da fatoração $\mathbf{LU}$, com ou sem pivoteamento em cada A_i, ou seja, resolvendo em cada processador o sistema

$$\mathbf{L}_i\mathbf{U}_i g = f \ . \tag{4.4}$$

O segundo sistema $Sx = g$ é fracamente acoplado, i.e., pode ser resolvido com pouca comunicação entre processadores. Podemos observar que resolver esse sistema é equivalente a resolver um sistema $\hat{S}\hat{x} = \hat{g}$ de menor complexidade, chamado de sistema reduzido. Esse sistema é composto pelas **k** linhas de S imediatamente acima e abaixo de cada divisão do particionamento, como podemos observar na Figura 4.3.

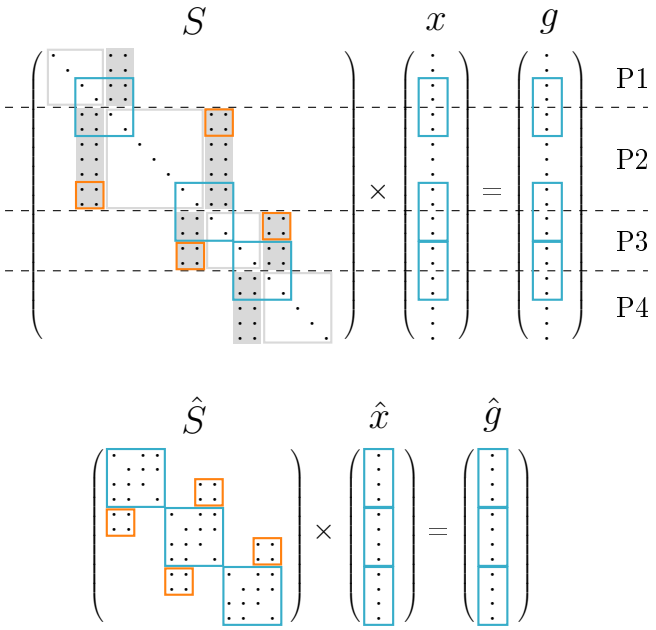


Figura 4.3: Esquematização do sistema reduzido $\hat{S}\hat{x} = \hat{g}$

Para resolver o sistema reduzido, considere V_i e W_i particionados como

$$V_i = \begin{pmatrix} V_i^{(t)} \\ V_i^{(m)} \\ V_i^{(b)} \end{pmatrix} \text{ e } W_i = \begin{pmatrix} W_i^{(t)} \\ W_i^{(m)} \\ W_i^{(b)} \end{pmatrix}. \tag{4.5}$$

Analogamente, xi e gi particionados como

$$x_i = \begin{pmatrix} x_i^{(t)} \\ x_i^{(m)} \\ x_i^{(b)} \end{pmatrix} \text{ e } g_i = \begin{pmatrix} g_i^{(t)} \\ g_i^{(m)} \\ g_i^{(b)} \end{pmatrix}. \tag{4.6}$$

onde $V_i^{(t)}$, $V_i^{(b)}$, $W_i^{(t)}$ e $W_i^{(b)}$ são as extremidades das matrizes *spikes* V_i e W_i, de tamanho $\mathbf{k} \times \mathbf{k}$. Da mesma forma, $x_i^{(t)}$, $x_i^{(b)}$ e $g_i^{(t)}$, $g_i^{(b)}$ representam as extremidades dos vetores x_i e g_i de tamanho **k**. Os sobrescritos (t), (m), e (b) são abreviações de *top* (cima), *middle* (meio) e *bottom* (baixo), respectivamente.

O sistema reduzido $\hat{S}\hat{x} = \hat{g}$ esquematizado na Figura 4.3 pode ser representado matricialmente, com $\hat{S}$ escrito na forma de uma matriz bloco tridiagonal.

$$\begin{pmatrix} I & V_1^{(b)} & & & & \\ W_2^{(t)} & I & & V_2^{(t)} & & \\ W_2^{(b)} & & I & V_2^{(b)} & & \\ & & W_3^{(t)} & I & & V_3^{(t)} \\ & & W_3^{(b)} & & I & V_3^{(b)} \\ & & & & W_4^{(t)} & I \end{pmatrix} \begin{pmatrix} \hat{x}_1^{(b)} \\ \hat{x}_2^{(t)} \\ \hat{x}_2^{(b)} \\ \hat{x}_3^{(t)} \\ \hat{x}_3^{(b)} \\ \hat{x}_4^{(t)} \end{pmatrix} = \begin{pmatrix} \hat{g}_1^{(b)} \\ \hat{g}_2^{(t)} \\ \hat{g}_2^{(b)} \\ \hat{g}_3^{(t)} \\ \hat{g}_3^{(b)} \\ \hat{g}_4^{(t)} \end{pmatrix}. \tag{4.7}$$

Uma vez obtida a solução $\hat{x}$ do sistema reduzido, representado na Equação 4.7, a solução x do sistema global pode ser obtida com perfeito paralelismo fazendo

$$\begin{cases} x_1 = g_1 - V_1\hat{x}_2^{(t)} \\ x_2 = g_2 - W_2\hat{x}_1^{(b)} - V_2\hat{x}_3^{(t)} \\ x_3 = g_3 - W_3\hat{x}_2^{(b)} - V_4\hat{x}_4^{(t)} \\ x_4 = g_4 - W_4\hat{x}_3^{(b)} \end{cases}. \tag{4.8}$$

E de forma geral,

$$\begin{cases} x_1 = g_1 - V_1\hat{x}_2^{(t)} \\ x_i = g_i - W_i\hat{x}_{i-1}^{(b)} - V_i\hat{x}_{i+1}^{(t)} \qquad i = 2, ..., p-1 \\ x_p = g_p - W_p\hat{x}_{p-1}^{(b)} \end{cases}. \tag{4.9}$$

A menos de erros de arredondamento, após a etapa de recuperação do valor de x, na Equação 4.9, temos o valor exato da solução do sistema $Ax = f$. Note que as contribuições $W_1\hat{x}_0^{(b)}$ (no processador 1) e $V_p\hat{x}_{p+1}^{(t)}$ (no processador p) não existem e, portanto, são consideradas nulas.

Entretanto, utilizando a abordagem descrita, a etapa de montagem do sistema reduzido $\hat{S}\hat{x} = \hat{g}$ pode se tornar muito cara computacionalmente. Isso porque as matrizes V e W são densas com quantidade de elementos igual a $n_i \times \mathbf{k}$ cada, o que demanda uma grande quantidade de memória para armazenamento. Com o objetivo de melhorar a eficiência do algoritmo, reduzindo os cálculos computacionais envolvidos na decomposição **LU** e o uso de memória, podemos calcular apenas as extremidades superiores e inferiores de tamanho $\mathbf{k} \times \mathbf{k}$ das matrizes V_i, W_i e do vetor g_i. Assim, a solução final do sistema $Ax = f$ pode ser recuperada fazendo

$$\begin{cases} \mathbf{L}_1\mathbf{U}_1x_1 = f_1 - B_1\hat{x}_2^{(t)} \\ \mathbf{L}_i\mathbf{U}_ix_i = f_i - C_i\hat{x}_{i-1}^{(b)} - B_i\hat{x}_{i+1}^{(t)} \qquad i = 2, ..., p-1 \\ \mathbf{L}_p\mathbf{U}_px_p = f_p - C_p\hat{x}_{p-1}^{(b)} \end{cases}. \tag{4.10}$$

Em suma, o SPIKE pode ser dividido nos seguintes passos principais:

1. Fatorar os blocos diagonais A_i em cada processador, independentes entre si.
2. Calcular as matrizes *spikes* V_i e W_i usando a fatoração obtida no passo anterior e calcular a solução do sistema $Dg = f$.
3. Montar e resolver o sistema reduzido $\hat{S}\hat{x} = \hat{g}$ e obter a solução parcial $\hat{x}$ imediatamente acima e abaixo de cada divisão do particionamento.
4. Recuperar o restante da solução de x.

4.2 Estratégias SPIKE

O algoritmo SPIKE original pode assumir diferentes estratégias durante cada uma de suas principais etapas. Quando se trata das variantes do SPIKE, o algoritmo é frequentemente referido como um *polyalgorithm*. Este termo, sem tradução direta para o português, é usado para classificar um conjunto de algoritmos que possuem regras que devem ser adequadamente escolhidas de acordo com as características do problema e da arquitetura paralela utilizada. Sendo assim,

1. Os blocos diagonais podem ser resolvidos:
   (a) Com método direto ($\mathbf{LU}, \mathbf{UL}$)
   (b) Com método iterativo
2. Os *spikes* podem ser calculados:
   (a) Explicitamente (totalmente ou parcialmente)
   (b) Implicitamente
3. O sistema reduzido por ser resolvido:
   (a) Com método direto (SPIKE Recursivo)
   (b) Com método iterativo precondicionado
   (c) Aproximadamente (SPIKE Truncado)

Neste trabalho, a estratégia conhecida como SPIKE Truncado — ou SPIKE-TU — foi escolhida com o objetivo de utilizar o algoritmo SPIKE como precondicionador de um método iterativo. Neste modelo, os blocos diagonais são calculados usando método direto de fatoração **LU** e **UL**, sem pivoteamento, as matrizes *spikes* são calculadas parcialmente (somente as extremidades), de forma explícita e exata, e o sistema reduzido é aproximado (truncado), resolvido por método direto.

4.3 Precondicionador SPIKE-TU

O SPIKE-TU ou SPIKE Truncado apresenta algumas mudanças com relação ao algoritmo clássico. Tais mudanças são essenciais para aprimorar o paralelismo e diminuir o número de operações de ponto flutuante.

A primeira mudança ocorre no tamanho dos blocos de acoplamento. O tamanho $\mathbf{k} \times \mathbf{k}$ desses blocos deve ser igual a largura de banda da matriz no algoritmo SPIKE original, garantindo que todos os elementos sejam agrupados. No SPIKE Truncado não é necessário calcular a solução exata do sistema precondicionado e, por esse motivo, podemos considerar um tamanho fixo para $\mathbf{k}$ bem menor que a largura de banda (denotado $\tilde{\mathbf{k}}$), para reduzir a quantidade de operações envolvidas na decomposição **LU**.

Outra mudança acontece na obtenção das matrizes *spikes* V_i e W_i. Considerando que as matrizes bloco diagonais A_i sejam diagonais dominante, podemos usar a fatoração **LU**, sem pivoteamento, para obter a extremidade inferior das matrizes V_i usando apenas os blocos inferiores, de dimensão $\tilde{\mathbf{k}} \times \tilde{\mathbf{k}}$, das matrizes **L** e **U**. Analogamente, podemos obter a extremidade superior das matrizes W_i usando a fatoração **UL**, sem pivoteamento, calculando apenas os blocos superiores, de dimensão $\tilde{\mathbf{k}} \times \tilde{\mathbf{k}}$, das matrizes **U** e **L**. Podemos ver na Figura 4.4 a esquematização dessa estratégia, chamada de **LU**/**UL**, para o cálculo da extremidade inferior de V_i usando somente os blocos inferiores de **L** e **U**.

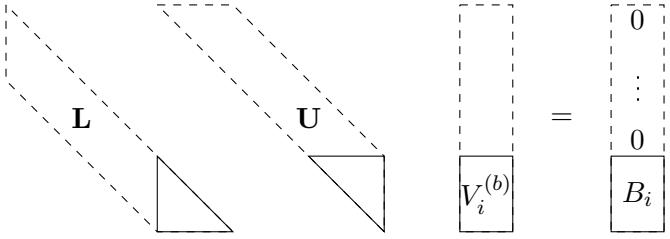


Figura 4.4: Estratégia **LU**/**UL** para V_i

Apesar dessa estratégia necessitar de mais uma fatoração, **UL**, Polizzi and Sameh [2006] afirmam, baseados em seus experimentos, que essa estratégia é mais eficiente do que usar apenas a fatoração **LU**, pois assim, o cálculo da matriz *spike* W_i completa, em cada bloco diagonal, seria evitado.

A próxima mudança no SPIKE-TU é com relação ao sistema reduzido. Baseado no estudo de Stephen Demko and Smith [1984], Polizzi and Sameh [2006] afirmam que se a matrix A é diagonal dominante, então a magnitude dos elementos das matrizes *spikes* V_i decaem de baixo para cima, enquanto que a magnitude dos elementos das matrizes *spikes* W_i decaem de cima para baixo. Assim, como o tamanho n dos blocos diagonais A_i é muito maior do que o tamanho $\tilde{\mathbf{k}}$ dos blocos B_i e C_i, a contribuição superior das matrizes *spikes* V_i e inferior das matrizes *spikes* W_i podem possuir valores aproximadamente zero e, portanto, serem desconsiderados. Além disso, como o algoritmo SPIKE-TU será usado como um precondicionador, não há necessidade da solução exata do sistema reduzido, portanto,

uma solução aproximada e mais simples de construir é suficiente. Neste caso, resolver o sistema reduzido truncado reduz a quantidade de comunicação e tem pouca influência na convergência do método iterativo. A Figura 4.5 mostra a esquematização da matriz S do sistema reduzido original e do sistema reduzido truncado, onde as contribuições $V_i^{(t)}$ e $W_i^{(b)}$ da Equação 4.7 destacados nos quadrados menores (em laranja) são descartadas.

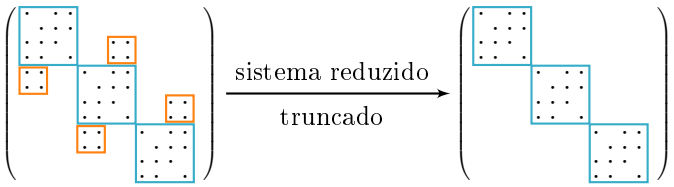


Figura 4.5: Esquematização da matriz S do sistema reduzido truncado

De maneira geral, cada sistema reduzido truncado $\tilde{S}_i$, destacados pelos quadrados maiores (em azul) da Figura 4.5, pode ser representado matricialmente como

$$\tilde{S}_i = \begin{pmatrix} I & V_i^{(b)} \\ W_{i+1}^{(t)} & I \end{pmatrix}. \tag{4.11}$$

Portanto, sendo p a quantidade total de processadores, o sistema reduzido truncado $\tilde{S} = diag\{\tilde{S}_1, \tilde{S}_2, ..., \tilde{S}_{p-1}\}$ será resolvido por partes em cada processador. Note que o sistema reduzido no processador i é formado pelas matrizes *spikes* $V_i^{(b)}$ e $W_{i+1}^{(t)}$ como mostrado na Equação 4.11. Em consequência disso, é necessário que todo processador $i+1$ envie a matriz $W_{i+1}^{(t)}$ para o processador i.

Para criar a matriz precondicionadora $M = \tilde{D} \times \tilde{S}$, é necessário aplicar um conjunto de reordenamentos na matriz A de forma que os blocos diagonais A_i e as matrizes bloco de acoplamento B_i e C_i contenham os elementos mais significativos da matriz, na tentativa de garantir um melhor precondicionador. Sendo assim, a matriz M é criada considerando os seguintes reordenamentos

$$M \approx KQPD_rAD_cP^TK^T, \tag{4.12}$$

onde D_r e D_c são fatores de *scaling*, P é uma permutação simétrica com o objetivo de reduzir a largura de banda da matriz, Q é uma permutação não simétrica para mover os maiores elementos para a diagonal e, por fim, K é uma permutação simétrica que move os elementos mais significativos para os blocos de acoplamento.

Esses reordenamentos são modelados por uma série de problemas combinatórios explicados detalhadamente no próximo capítulo. É importante ressaltar que nem sempre a aplicação de todas as estratégias combinatórias garantirão um melhor precondicionador. Na prática, é necessário conhecer as características do problema e testar a melhor combinação de estratégias.

Capítulo 5

Estratégias Combinatórias

A matriz precondicionadora M, criada a partir da matriz A, precisa ser transformada em uma matriz com propriedades específicas para que ela possa gerar um sistema equivalente ao original e ser resolvida pelo algoritmo SPIKE. Essas transformações são feitas aplicando um conjunto de permutações de linhas e/ou colunas e devem ser realizadas antes da etapa de pré-processamento do SPIKE.

Cada transformação aplicada na matriz original pode ser abordada como problemas combinatórios em grafos, por exemplo: *matching* perfeito em grafos, particionamento e rerrotulação de vértices de um grafo. Dentre os objetivos que desejamos alcançar estão:

1. garantir a não-singularidade de cada bloco diagonal.
2. obter uma matriz com estrutura de banda.
3. obter as matrizes bloco diagonais e de blocos de acoplamento.
4. mover elementos mais significativos para os blocos de acoplamento.

Estes objetivos são extremamente importantes para a construção de um bom precondicionador. Além disso, algumas estratégias combinatórias são indispensáveis para que o precondicionador SPIKE funcione corretamente.

Garantir a não-singularidade de cada bloco diagonal A_i é essencial, uma vez que, em algumas etapas do algoritmo SPIKE, esses blocos serão resolvidos através do método direto da fatoração **LU**. Obter uma matriz com estrutura de banda também é extremamente importante para o algoritmo SPIKE, que foi desenvolvido com essa premissa. Em seguida, obter as matrizes bloco diagonais e de blocos de acoplamento nada mais é do que realizar um particionamento dos dados, indispensável na programação paralela. Por fim, mover elementos mais significativos para os blocos de acoplamento é uma otimização importante que reforça o objetivo do precondicionador.

Nas próximas seções discutiremos cada uma dessas estratégias combinatórias, explicando seus principais objetivos no contexto do precondicionador SPIKE, os algoritmos utilizados, detalhes de suas implementações e exemplos numéricos.

5.1 *Matching* e *Scaling*

Na etapa de fatoração do algoritmo SPIKE, mais especificamente na Equação 4.2, os blocos diagonais A_i precisam ser resolvidos via método direto usando a fatoração **LU**. Para isso, é necessário que esses blocos sejam não-singulares, ou seja, admitam uma inversa. Esse objetivo pode ser interpretado como um *matching* perfeito em grafos capaz de mover os elementos mais significativos para a diagonal principal.

Seja $A = \{a_{ij}\}$ uma matriz esparsa geral e o grafo bipartido $G_A = (V_r, V_c, E)$ seu grafo associado, sendo V_r e V_c seus conjuntos disjuntos de vértices e seu conjunto de arestas E, onde $(u, v) \in E$ implica que $u \in V_r$ e $v \in V_c$. A aresta $(i, j) \in E$ se, e somente se, $a_{ij} \neq 0$. Assim, um subconjunto de arestas $M \subseteq E$ é um *matching* se não há duas arestas de M incidentes no mesmo vértice. Um *matching* maximal é perfeito, se cada vértice é incidente a uma aresta do *matching*. Segundo Duff and Koster [2000], apesar de nem todo grafo bipartido admitir um *matching* perfeito, esse *matching* existe se a matriz associada for não-singular. Portanto, um *matching* perfeito M tem cardinalidade n e define uma matriz de permutação $Q = \{q_{ij}\}$ como

$$\begin{cases} q_{ji} = 1, & \text{se } (i,j) \in M \\ q_{ji} = 0, & \text{caso contrário} \end{cases} , \tag{5.1}$$

onde a matriz QA — aplicada a permutação não simétrica de linhas Q — possui a diagonal principal livre de elementos nulos. Vale lembrar que, após a aplicação da permutação não simétrica, as matrizes simétricas perdem essa propriedade. Sendo assim, o novo sistema linear resultante é da forma

$$QAx = Qf \ . \tag{5.2}$$

Os algoritmos mais eficientes para encontrar *matchings* maximais em um grafo bipartido estão descritos em Duff and Koster [1999, 2000] e estão presentes na biblioteca HSL MC64[1]. Esta biblioteca, utilizada neste trabalho, implementa o algoritmo *matching* bipartido valorado (*weighted bipartite matching*) cujo objetivo é encontrar um conjunto de entradas, que não estejam na mesma linha e coluna, de forma que o produto ou a soma dessas entradas sejam maximizadas. O algoritmo inicia com um *matching* parcial e tenta estendê-lo procurando, no grafo bipartido correspondente, o menor caminho possível partindo de um vértice $u \in V_r$ para qualquer vértice $v \in V_c$, tais que u e v

[1] http://www.hsl.rl.ac.uk/catalogue/mc64.html

não compõem nenhuma aresta do *matching* corrente. Esses caminhos, chamados de *augmenting path* ou caminhos aumentados, são obtidos usando um algoritmo similar ao Dijkstra [Cormen et al., 2009].

A Figura 5.1 ilustra um grafo bipartido de tamanho 6, em (a), associado a uma matriz, em (b). Um *matching* valorado perfeito é computado. As arestas desse *matching* são mostradas em (c) e as respectivas entradas destacadas, em azul, na matriz em (d).

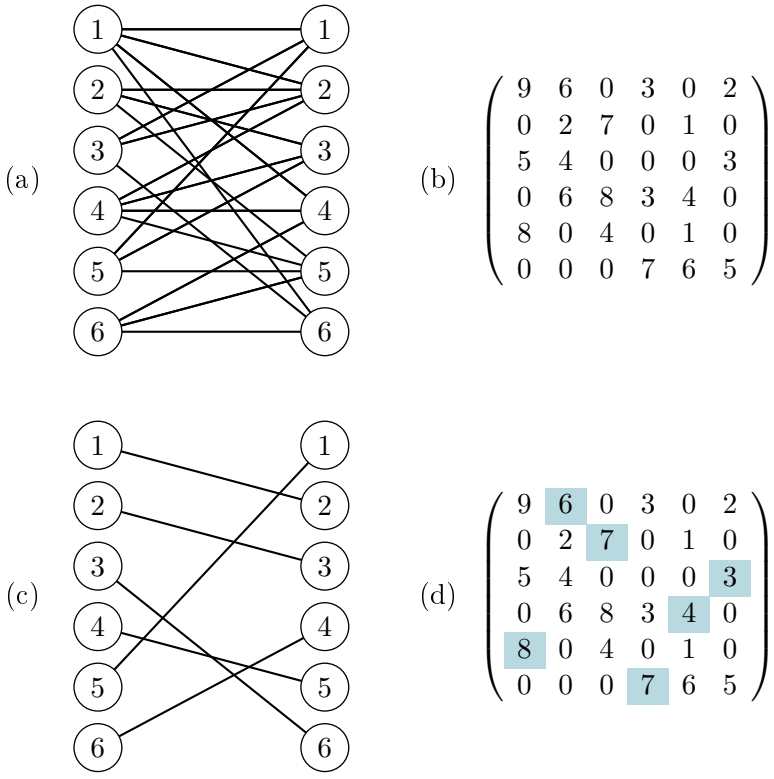


Figura 5.1: Exemplo de funcionamento do algoritmo *weighted bipartite matching*

Após a execução do *matching* na Figura 5.1(d), podemos extrair a permutação de linha Q = (5 1 2 6 4 3), que aplicada na matriz original, gera a matriz resultante

$$\begin{pmatrix} 8 & 0 & 4 & 0 & 1 & 0 \\ 9 & 6 & 0 & 3 & 0 & 2 \\ 0 & 2 & 7 & 0 & 1 & 0 \\ 0 & 0 & 0 & 7 & 6 & 5 \\ 0 & 6 & 8 & 3 & 4 & 0 \\ 5 & 4 & 0 & 0 & 0 & 3 \end{pmatrix}.$$

Na Figura 5.2 podemos observar o efeito do *matching* aplicado na matriz `gemat12`. A matriz original (esquerda) possui zeros em grande parte da diagonal principal e, após o *matching* (direita), toda a diagonal principal é preenchida por não nulos.

O algoritmo `MC64` de cálculo do *matching* também é capaz de calcular fatores que tornam a matriz normalizada. A técnica, chamada de *scaling*, consiste em multiplicar a

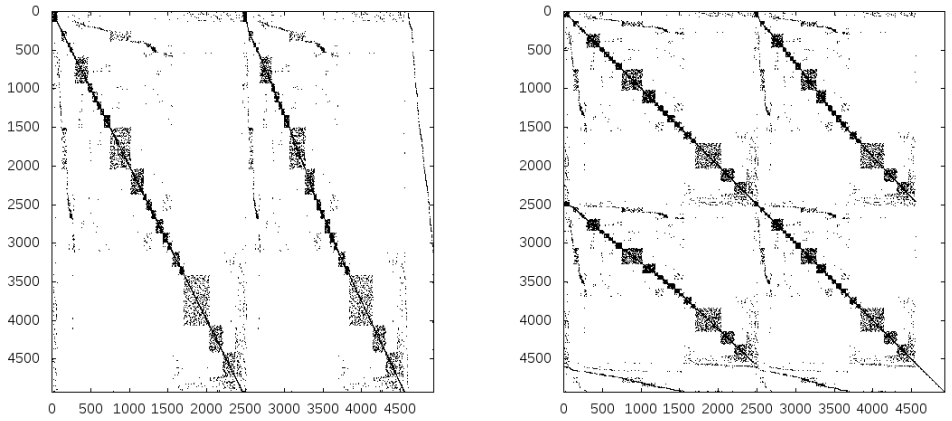


Figura 5.2: *Matching* aplicado na matriz `gemat12`

matriz por fatores de linha e coluna de tal forma que as entradas não nulas da diagonal principal assumam valores absolutos iguais a 1.0 e as demais, valores absolutos menores ou iguais a 1.0. Para esse fim, considere os logaritmos naturais dos fatores do *scaling* $D_r = exp\{u_1, u_2, u_3, ..., u_n\}$ para as linhas, e $D_c = exp\{v_1, v_2, v_3, ..., v_n\}$ para as colunas. Assim, o novo sistema após o *scaling* será

$$D_r A D_c^T D_c x = D_r f \ , \tag{5.3}$$

onde cada elemento não nulo da matriz A sofre a influência

$$a_{ij} = a_{ij} * \exp(u_i + v_j) \ . \tag{5.4}$$

Segundo Naumann and Schenk [2012], o *scaling* tem grande influência na escalabilidade e robustez do SPIKE, quando usado como precondicionador. A aplicação do *matching* e, em seguida, dos fatores D_r e D_c do *scaling*, na matriz exemplo é mostrada a seguir.

$$\begin{pmatrix} 8 & 0 & 4 & 0 & 1 & 0 \\ 9 & 6 & 0 & 3 & 0 & 2 \\ 0 & 2 & 7 & 0 & 1 & 0 \\ 0 & 0 & 0 & 7 & 6 & 5 \\ 0 & 6 & 8 & 3 & 4 & 0 \\ 5 & 4 & 0 & 0 & 0 & 3 \end{pmatrix} \rightarrow \begin{pmatrix} 1.00 & 0.00 & 0.56 & 0.00 & 0.28 & 0.00 \\ 1.00 & 1.00 & 0.00 & 0.64 & 0.00 & 0.60 \\ 0.00 & 0.38 & 1.00 & 0.00 & 0.29 & 0.00 \\ 0.00 & 0.00 & 0.00 & 1.00 & 1.00 & 1.00 \\ 0.00 & 1.00 & 1.00 & 0.64 & 1.00 & 0.00 \\ 0.62 & 0.74 & 0.00 & 0.00 & 0.00 & 1.00 \end{pmatrix} .$$

5.2 Reordenamento

Para que o algoritmo SPIKE possa ser aplicado para uma matriz geral, devemos primeiramente transformá-la em uma matriz com estrutura de banda através de um reordenamento que minimize sua largura de banda. Isso porque o SPIKE necessita que todos os elementos não nulos estejam cobertos pelas matrizes bloco diagonais A_i e pelas matrizes bloco de acoplamento B_i e C_i. Para que todos os elementos da matriz sejam

cobertos, as matrizes B_i e C_i precisam ter uma dimensão mínima **k**, onde **k** é o valor da largura de banda da matriz. Reduzindo a largura de banda, e consequentemente o valor de **k**, mais eficiente se torna o processo de encontrar as matrizes V_i e W_i, uma vez que a complexidade dos sistemas na Equação 4.2 é reduzida.

A minimização da largura de banda de matrizes esparsas é um problema $\mathcal{NP}$-Completo [Papadimitriou, 1976]. Algoritmos com tal objetivo foram bastante estudados na últimas décadas e continuam apresentando grande vantagem computacional em diversas aplicações. Por exemplo, Benzi et al. [1999] estudaram detalhadamente a redução do preenchimento (*fill-in*) causado pela fatoração **LU** Incompleta, ou ILU. Ghidetti [2011] e Lugon [2013] aplicaram a mesma ideia para comparar um vasto conjunto de algoritmos de reordenamento, tais como: *Reverse Cuthill-McKee* (RCM) [Cuthill and McKee, 1969], algoritmo *Sloan* [Sloan, 1986], algoritmo Espectral [Barnard et al., 1993], *Approximate Minimum Degree* (AMD) [Davis et al., 1994], *Nested Dissection* (ND) [George, 1973] e *Gibbs-Poole-Stockmeyer* (GPS) [Gibbs et al., 1976].

Neste trabalho, utilizamos o algoritmo Espectral proposto por Barnard et al. [1993]. Dada uma matriz A e seu grafo associado G, Fiedler [1973] diz que o autovetor v associado ao 2^o menor autovalor λ_2 da matriz Laplaciana de G define sua conectividade algébrica. A maior dificuldade do algoritmo é encontrar esse vetor, também conhecido como vetor de *Fiedler*. Para isto, utilizamos uma implementação serial contida na biblioteca HSL MC73[2] que calcula uma aproximação desse vetor, com uma abordagem multinível, usando o algoritmo SYMMLQ para resolver os sistemas lineares dentro do método *Rayleigh Quotient Iterations* (RQI). Ao fim da execução dessa estratégia combinatória, temos uma matriz de permutação simétrica P que minimiza a largura de banda da matriz original e deve ser aplicada em todo o sistema linear de acordo com

$$PAP^T Px = Pf \ . \tag{5.5}$$

Na Figura 5.3 podemos observar o efeito do reordenamento Espectral aplicado na matriz `gemat12` após a aplicação do *matching*, mostrado na Figura 5.2. Como podemos ver, houve uma grande redução da largura de banda dessa matriz que, inicialmente, era de 38.312 (esquerda) e passou para 702 (direita).

Um estudo mais recente de Manguoglu et al. [2010] sugere uma modificação do algoritmo Espectral, chamado de Espectral Valorado (*weighted spectral*), que considera a magnitude dos elementos não nulos, tentando mover as entradas mais significativas para mais próximo da diagonal principal. O estudo apresentou bons resultados deste reordenamento em testes feitos especificamente no precondicionador SPIKE. As entradas mais significativas, por estarem mais próximas da diagonal principal, serão englobadas pelo precondicionador, enquanto que os elementos de menor valor absoluto poderão ser descartados.

[2] http://www.hsl.rl.ac.uk/catalogue/hsl_mc73.html

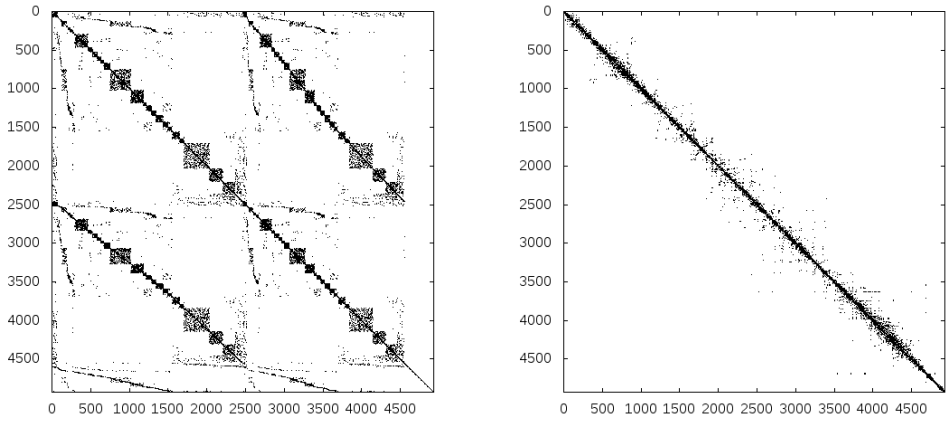


Figura 5.3: Reordenamento Espectral aplicado na matriz `gemat12`

A mudança com relação ao algoritmo original ocorre apenas na definição da matriz Laplaciana. Neste caso, utiliza-se a definição da Laplaciana valorada $\bar{L}$ dada por

$$\bar{L}_{ij} = \begin{cases} \sum_j |a_{ij}| & \text{se } i = j \\ -|a_{ij}| & \text{se } i \neq j \end{cases} . \tag{5.6}$$

O Algoritmo 1 exibe o pseudocódigo do reordenamento Espectral Valorado da biblioteca `HSL MC73`, cujos testes serão analisados neste trabalho.

Algoritmo 1: Espectral

1 Calcular o 2^o menor autovalor da matriz Laplaciana $\bar{L}$ e o autovetor v associado
2 Ordenar v em ordem crescente (ou decrescente)

5.3 Particionamento

O objetivo principal do particionamento, além de reduzir a comunicação entre os processadores, é garantir que cada partição tenha uma quantidade balanceada de trabalho a ser realizado que, neste caso, é garantir que cada partição tenha, aproximadamente, a mesma quantidade de elementos não nulos. Como o tempo de execução de um programa paralelo é medido em função do processo mais lento, equilibrar a quantidade de elementos não nulos é assegurar que cada nó distribuído faça, aproximadamente, a mesma quantidade de operações de ponto flutuante.

O problema de particionamento consiste em encontrar as matrizes bloco diagonais A_i e as matrizes bloco de acoplamento C_i e B_i. Várias heurísticas foram propostas para resolver o problema geral de particionamento em grafos, sendo as mais conhecidas *Kernighan–Lin* [Kernighan and Lin, 1970] e *Fiduccia-Mattheyses* [Fiduccia and Mattheyses, 1982].

Neste trabalho, o particionamento foi modelado como um problema específico, chamado de *Chains-on-chains partitioning* (CCP) [Pinar and Aykanat, 2004] que, aplicado após o reordenamento precisa considerar posições contíguas na matriz. O objetivo é encontrar uma sequência de $d-1$ índices separadores para dividir uma cadeia de tarefas com pesos computacionais associados em d partes consecutivas, minimizando a carga da parte mais carregada. Na nossa aplicação, as tarefas representam as n linhas da matriz, seus pesos são as quantidades de elementos não-nulos nas linhas e o efeito do particionamento é distribuir os elementos não-nulos de forma equilibrada entre os processadores. O CCP pode ser resolvido em tempo polinomial, e o algoritmo *MinMax* [Manne and Sørevik, 1995] é usado para obter a solução exata deste problema.

A Figura 5.4 mostra um particionamento para quatro processadores da matriz `gemat12`, após a aplicação do *matching* e do reordenamento Espectral.

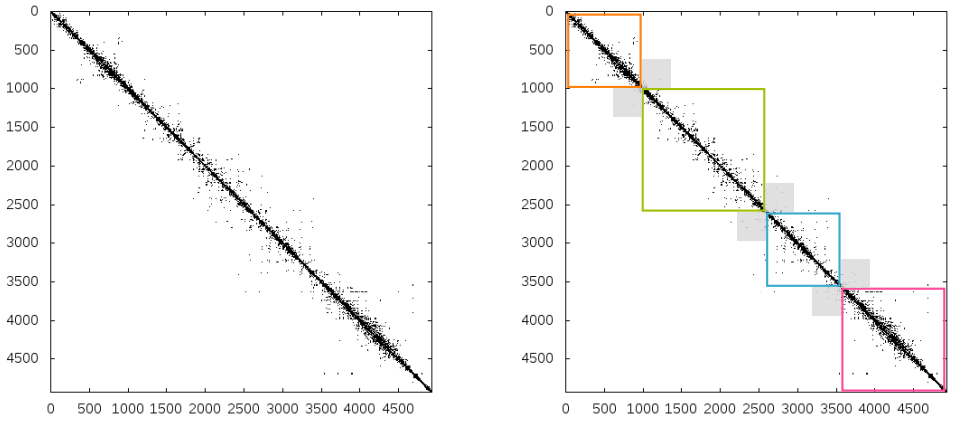


Figura 5.4: Particionamento aplicado na matriz `gemat12`

5.4 Problema Quadrático da Mochila (PQM)

Nesta estratégia, o objetivo é aplicar uma permutação simétrica K para concentrar elementos mais significativos, em módulo, dentro das matrizes bloco de acoplamento B_i e C_i de tamanho $\mathbf{k} \times \mathbf{k}$. Para esse fim, utilizamos a heurística *DeMin*, proposta por Naumann and Schenk [2012], que encontra uma permutação tal que o novo sistema será

$$KAK^T Kx = Kf \ . \tag{5.7}$$

Considere a matriz $N_{i,i+1} = |A_{i,i+1}| + |A^T_{i+1,i}|$, i.e, N é a soma, em módulo, da submatriz formada pelas linhas do bloco A_i e colunas do bloco A_{i+1} com a transposta da submatriz formada pelas colunas do bloco A_i e linhas do bloco A_{i+1}. O objetivo do algoritmo *DeMin* é obter a submatriz quadrada $\mathbf{k} \times \mathbf{k}$ mais densa de N. As linhas e colunas dessa submatriz $\mathbf{k} \times \mathbf{k}$ determinam uma permutação K a ser aplicada na matriz para alinhar as matrizes bloco de acoplamento junto às matrizes bloco diagonais.

5.5 Exemplo Numérico

As próximas figuras mostram um exemplo numérico da influência de cada uma das estratégias combinatórias descritas. Considere a matriz não simétrica em sua configuração original, gerada aleatoriamente, mostrada na Figura 5.5. Esta matriz possui dimensão n igual a 15 e quantidade de elementos não nulos igual a 59.

$$\begin{pmatrix}
 & & & & 8.10 & 6.58 & 1.70 & & & & 3.80 & & & \\
 & & & & & 7.29 & 2.87 & & 9.24 & & & & 0.31 & \\
1.22 & & & & 9.79 & & & 5.78 & & 5.52 & & & & \\
2.75 & & & & & & 3.59 & & & 6.53 & & & & \\
 & 6.53 & & & & & & & & & & & & 4.09 \\
3.97 & & 8.79 & 3.90 & 0.61 & & & & & 7.23 & & 4.09 & & 5.02 \\
 & & 9.49 & 4.15 & 1.96 & & & & & & 9.92 & & & 6.14 \\
4.82 & & 0.50 & & 2.94 & 9.84 & & & & & & & & \\
 & & & 5.40 & 3.04 & & & 6.89 & & & & & 8.88 & \\
 & 7.44 & 1.81 & 6.91 & & & & & & & & & & \\
 & & & & & & & 7.37 & & & 1.52 & & & 7.44 \\
 & & 2.12 & & 4.07 & & & & & & & & 9.30 & 8.71 \\
 & & & & 5.21 & 0.74 & & & 2.53 & & & & & \\
5.77 & & & & & & & 8.82 & 3.88 & 8.47 & 2.45 & & & \\
 & & & & & & 4.41 & & 4.78 & & & 6.70 & & 3.80 & \\
\end{pmatrix}$$

Figura 5.5: Exemplo Numérico: Matriz original

A Figura 5.6 mostra a matriz após a aplicação do *matching* e do *scaling*. Podemos perceber que a permutação não simétrica, encontrada pelo *matching*, foi capaz de preencher toda a diagonal principal. Além disso, os fatores do *scaling* transformaram todos os elementos menores ou iguais a 1.0, em módulo.

$$\begin{pmatrix}
1.00 & & & & & & & 1.00 & 0.53 & 1.00 & 0.36 & & & \\
 & 1.00 & & & & & & & & & & & & 0.48 \\
0.53 & & 1.00 & 0.48 & 0.03 & & & & & 0.66 & & 0.46 & & 0.43 \\
 & 1.00 & 0.24 & 1.00 & & & & & & & & & & \\
 & & & & 1.00 & 0.18 & & & 1.00 & & & & & \\
1.00 & & 0.09 & & 0.23 & 1.00 & & & & & & & & \\
0.60 & & & & & & 1.00 & & & 0.96 & & & & \\
 & & & & & & & 1.00 & & & 0.27 & & & 1.00 \\
 & & & & & 0.49 & 0.51 & & 1.00 & & & & 0.06 & \\
0.32 & & & & 1.00 & & & 1.00 & & 1.00 & & & & \\
 & & 0.96 & 0.45 & 0.09 & & & & & & 1.00 & & & 0.47 \\
 & & & & 0.96 & 1.00 & 0.68 & & & & & 1.00 & & \\
 & & & 0.92 & 0.22 & & & 0.84 & & & & & 1.00 & \\
 & & & & & & 1.00 & & 0.67 & & & 1.00 & & 1.00 \\
 & & 0.32 & & 0.28 & & & & & & & & 1.00 & & 1.00 \\
\end{pmatrix}$$

Figura 5.6: Exemplo Numérico: *matching* e *scaling*

A próxima estratégia combinatória realiza uma permutação simétrica com o objetivo de transformar a matriz original em uma matriz com largura de banda estreita, condição necessária para a utilização do algoritmo SPIKE. O reordenamento Espectral, mostrado na Figura 5.7, foi capaz de reduzir a largura de banda da matriz original de 12 para 5.

$$\begin{pmatrix}
1.00 & & & 0.48 & & & & & & & & & & & \\
1.00 & 1.00 & & & & & 0.24 & & & & & & & & \\
 & 0.92 & 1.00 & & & 0.84 & & 0.22 & & & & & & & \\
 & & 1.00 & 1.00 & & & 0.32 & 0.28 & & & & & & & \\
 & 0.45 & & 0.47 & 1.00 & & 0.96 & 0.09 & & & & & & & \\
 & & & 1.00 & 0.27 & 1.00 & & & & & & & & & \\
 & 0.48 & & 0.43 & & & 1.00 & 0.03 & 0.66 & 0.53 & & 0.46 & & & \\
 & & & & & & & 1.00 & & & 0.18 & & 1.00 & & \\
 & & & & & 1.00 & & 1.00 & 1.00 & 0.32 & & & & & \\
 & & & & 0.36 & 1.00 & & & 1.00 & 1.00 & & & 0.53 & & \\
 & & & & & & 0.09 & 0.23 & & 1.00 & 1.00 & & & & \\
 & & & & & & & 0.96 & & & 1.00 & 1.00 & & 0.68 & \\
 & & & & & & & & & & 0.49 & & 1.00 & 0.51 & 0.06 \\
 & & & & & & & & 0.96 & 0.60 & & & & 1.00 & \\
 & & & & & & & & & & & 1.00 & 0.67 & 1.00 & 1.00
\end{pmatrix}$$

Figura 5.7: Exemplo Numérico: reordenamento Espectral

O particionamento não altera os coeficientes da matriz do estágio anterior, especificamente. Entretanto, esta estratégia é necessária para encontrar as matrizes bloco diagonais e as matrizes bloco de acoplamento. A Figura 5.8 mostra um particionamento realizado para três processadores e com dimensão **k** das matrizes bloco de acoplamento B_i e C_i igual a 2.

$$\begin{pmatrix}
1.00 & & & 0.48 & & & & & & & & & & & \\
1.00 & 1.00 & & & & & 0.24 & & & & & & & & \\
 & 0.92 & 1.00 & & & 0.84 & & 0.22 & & & & & & & \\
 & & 1.00 & 1.00 & & & 0.32 & 0.28 & & & & & & & \\
 & 0.45 & & 0.47 & 1.00 & & 0.96 & 0.09 & & & & & & & \\
 & & & 1.00 & 0.27 & 1.00 & & & & & & & & & \\
 & 0.48 & & 0.43 & & & 1.00 & 0.03 & 0.66 & 0.53 & & 0.46 & & & \\
 & & & & & & & 1.00 & & & 0.18 & & 1.00 & & \\
 & & & & & 1.00 & & 1.00 & 1.00 & 0.32 & & & & & \\
 & & & & 0.36 & 1.00 & & & 1.00 & 1.00 & & & 0.53 & & \\
 & & & & & & 0.09 & 0.23 & & 1.00 & 1.00 & & & & \\
 & & & & & & & 0.96 & & & 1.00 & 1.00 & & 0.68 & \\
 & & & & & & & & & & 0.49 & & 1.00 & 0.51 & 0.06 \\
 & & & & & & & & 0.96 & 0.60 & & & & 1.00 & \\
 & & & & & & & & & & & 1.00 & 0.67 & 1.00 & 1.00
\end{pmatrix}$$

Figura 5.8: Exemplo Numérico: particionamento

A estratégia mostrada na Figura 5.9 tenta mover a maior quantidade de elementos (e os mais significativos) para dentro dos blocos de acoplamento. Como podemos ver, após a aplicação do Problema Quadrático da Mochila, sete elementos foram movidos para dentro dos blocos de acoplamento (área cinza), enquanto que na configuração anterior, apenas três elementos estavam dentro desses blocos.

$$
\begin{pmatrix}
1.00 & & & 0.48 & & & & & & & & & & & \\
1.00 & 1.00 & & & & & & 0.24 & & & & & & & \\
 & 0.92 & 1.00 & & & 0.84 & & & & 0.22 & & & & & \\
 & & 1.00 & 1.00 & & & & 0.32 & & 0.28 & & & & & \\
 & 0.45 & & 0.47 & 1.00 & & & 0.96 & & 0.09 & & & & & \\
 & & & 1.00 & 0.27 & 1.00 & & & & & & & & & \\
 & & & & & 1.00 & 1.00 & & 0.32 & 1.00 & & & & & \\
 & 0.48 & & 0.43 & & & 0.66 & 1.00 & 0.53 & 0.03 & & & 0.46 & & \\
 & & & & 0.36 & 1.00 & 1.00 & & 1.00 & & & 0.53 & & & \\
 & & & & & & & & & 1.00 & 0.18 & 1.00 & & & \\
 & & & & & & & 0.09 & 1.00 & 0.23 & 1.00 & & & & \\
 & & & & & & & & & & 0.49 & 1.00 & & 0.51 & 0.06 \\
 & & & & & & & & & 0.96 & 1.00 & & 1.00 & 0.68 & \\
 & & & & & & 0.96 & & 0.60 & & & & & 1.00 & \\
 & & & & & & & & & & & 0.67 & 1.00 & 1.00 & 1.00
\end{pmatrix}
$$

Figura 5.9: Exemplo Numérico: Problema Quadrático da Mochila

Capítulo 6

Implementações

6.1 SPIKE

A implementação do algoritmo PSPIKE foi baseada nos passos descritos em Naumann and Schenk [2012]. O Algoritmo 2 descreve esses passos:

Algoritmo 2: PSPIKE

1. Ler a matriz A e armazenar em uma estrutura de dados otimizada.
2. Aplicar as estratégias combinatórias na matriz A, criando a matriz precondicionadora $M = \tilde{D} \times \tilde{S} \approx KQPD_rAD_cP^TK^T$, onde D_r e D_c são fatores de *scaling*, P é uma permutação simétrica com o objetivo de reduzir a largura de banda da matriz, Q é uma permutação não simétrica para mover os maiores elementos para a diagonal e K é uma permutação simétrica para mover os elementos mais significativos para os blocos de acoplamento.
3. Particionar a matriz A e enviar as matrizes C_i, A_i e B_i para o processador i. *{O processador* 0 *calcula apenas* B_0 *e o último processador, p, calcula apenas* C_p*}*.
4. Aplicar a fatoração **LU** nas matrizes A_i usando o PARDISO em paralelo.
5. Resolver com o PARDISO: $\mathbf{L}_i\mathbf{U}_i[V_i^{(b)}, W_i^{(t)}] = \left[\binom{0}{B_i}, \binom{C_i}{0}\right]$. *{O processador* 0 *calcula apenas* $V_0^{(b)}$ *e o processador p apenas* $W_p^{(t)}$*}*.
6. Enviar a matriz $W_i^{(t)}$ de tamanho $\tilde{\mathbf{k}}$ para o processador $i-1$ e montar o sistema reduzido truncado $\tilde{S}_i$ da Equação 4.11 de tamanho $2\tilde{\mathbf{k}}$. *{O processador* 0 *somente recebe e o último processador, p, somente envia}*.
7. Aplicar a fatoração **LU** no sistema reduzido truncado $\tilde{S}_i$.
8. Recuperar o vetor de termos independentes $f^r = KQPD_rf$.
9. Aplicar o algoritmo GMRES distribuído realizando, a cada iteração, um produto matriz-vetor eficiente e a operação de precondicionamento $M\bar{x} = z$ (Algoritmo 3).
10. Recuperar a solução x^r do sistema precondicionado, reordenando a solução final do sistema linear fazendo $x = D_cP^TK^Tx^r$.

No passo 9 do Algoritmo 2 é necessário resolver um produto matriz-vetor (detalhado na próxima seção) e, em seguida, realizar a operação de precondicionamento $M\bar{x} = z$. O Algoritmo 3 mostra o passo a passo de como esta etapa é feita no algoritmo PSPIKE.

Algoritmo 3: Etapas da operação $M\bar{x} = z$ do GMRES precondicionado

1 Resolver com o PARDISO $\mathbf{L}_i\mathbf{U}_i g_i = z_i$.
2 Particionar $g_i = (g_i^{(b)}, g_i^{(m)}, g_i^{(t)})$, sendo as partes inferiores e superiores de tamanho $\tilde{\mathbf{k}}$, e enviar a parte superior $g_i^{(t)}$ para o processador anterior, $i-1$. *{O processador 0 somente recebe e o último processador, p, somente envia}.*
3 Resolver o sistema reduzido truncado $\begin{pmatrix} I & V_i^{(b)} \\ W_{i+1}^{(t)} & I \end{pmatrix} \begin{pmatrix} \bar{x}_i^{(b)} \\ \bar{x}_{i+1}^{(t)} \end{pmatrix} = \begin{pmatrix} \bar{g}_i^{(b)} \\ \bar{g}_{i+1}^{(t)} \end{pmatrix}$. *{O último processador, p, fica ocioso}.*
4 Enviar a solução $\bar{x}_i^{(b)}$ para o processador $i+1$. *{O processador p somente recebe}.*
5 Resolver com o PARDISO $\mathbf{L}_i\mathbf{U}_i\bar{x}_i = z_i - B_i\bar{x}_{i+1}^{(t)} - C_i\bar{x}_{i-1}^{(b)}$. *{O processador 0 não possui $\bar{x}_{i-1}^{(b)}$, e o processador p, não possui $\bar{x}_{i+1}^{(t)}$. Portanto, considere-os zero.}*

6.2 Produto matriz-vetor paralelo

A operação de multiplicação de uma matriz esparsa por um vetor, comumente abreviado por SpMV (*Sparse matrix-vector multiplication*), é a operação mais importante realizada dentro de um método iterativo. Esta operação representa a porção de tempo mais significativa desses métodos e, por esse motivo, precisa ser implementada de forma eficiente. Várias técnicas podem ser empregadas para alcançar essa eficiência, tais como o uso de estruturas de dados especiais para tipos específicos de matrizes ou a utilização de técnicas da programação paralela em sistemas com multiprocessadores. Diversos estudos e técnicas foram propostos para alcançar um bom desempenho em implementações paralelas para o SpMV, incluindo a utilização de abordagens híbridas usando MPI e OpenMP [Schubert et al., 2011, Ye et al., 2015].

O principal desafio na realização do SpMV é encontrar uma maneira de distribuir a matriz e o vetor entre os processadores buscando o melhor desempenho computacional, i.e., reduzir os gastos com a comunicação entre processadores. Apresentamos nas próximas seções, os três principais esquemas que podem ser utilizados para realizar a decomposição dos dados da matriz e do vetor entre os processadores.

6.2.1 Decomposição 1D por Linhas

No esquema de decomposição unidimensional por linhas, mostrado na Figura 6.1 com quatro processadores, cada processador possui um bloco de linhas da matriz A e, ao final da operação, possuirão uma porção do vetor de saída y (ou solução). Nesse modelo, o vetor x precisa ser conhecido por completo por todos os processadores, portanto, é necessário comunicá-lo através de uma operação de *broadcast*. No caso da programação da decomposição 1D por linhas usando memória compartilhada, a leitura do vetor x está sujeita a acessos concorrentes.

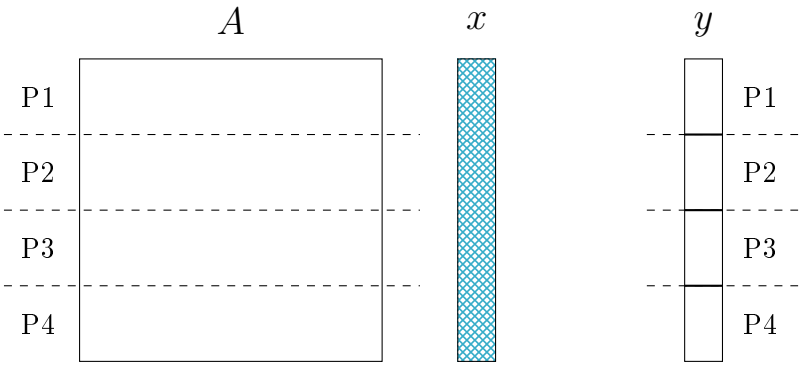


Figura 6.1: Decomposição 1D por linhas para o SpMV

6.2.2 Decomposição 1D por Colunas

No esquema de decomposição unidimensional por colunas, cada processador possui um bloco de colunas da matriz A e a porção equivalente do vetor de entrada x. Ao final da operação, cada processador terá calculado uma parte da solução de todo o vetor de saída y. Portanto, é necessário que haja uma sincronização para que os resultados de cada processador sejam somados à solução final. Diferente do modelo anterior, a decomposição 1D por colunas está sujeita a escritas concorrentes no caso da programação usando memória compartilhada. A Figura 6.2 mostra o esquema para quatro processadores.

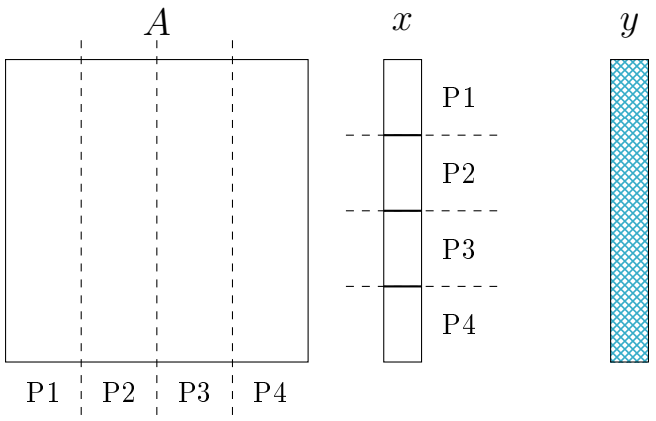


Figura 6.2: Decomposição 1D por colunas para o SpMV

6.2.3 Decomposição 2D

A decomposição bidimensional, representada na Figura 6.3 para 16 processadores, divide a matriz A em blocos de linhas e de colunas de modo que cada bloco é distribuído para um processador, assim como os vetores de entrada x e de saída y. Neste caso, se for utilizado memória compartilhada, poderão ocorrer concorrências no acesso à memória tanto durante a leitura (no vetor x) como durante a escrita (no vetor y).

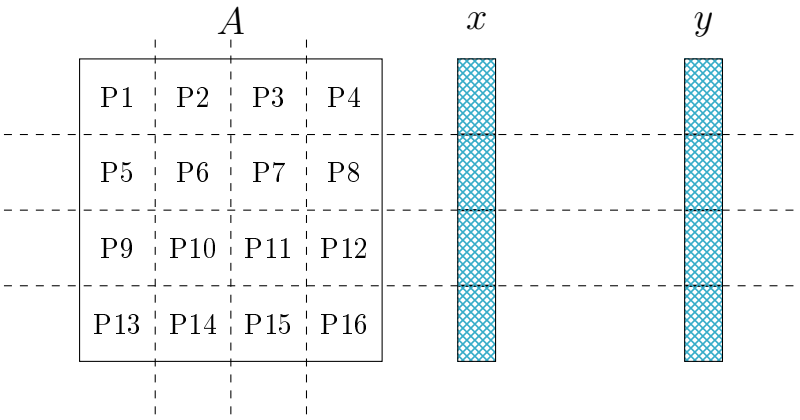


Figura 6.3: Decomposição 2D para o SpMV

O esquema de decomposição 2D é um modelo simples e pode ser facilmente implementado utilizando programação paralela distribuída ou compartilhada. No contexto deste trabalho, decidimos tirar proveito da partição realizada pelo algoritmo SPIKE e, portanto, foi proposto um modelo baseado na decomposição 2D.

Assumindo que o SPIKE necessita de uma matriz com estrutura de banda, a distribuição dos dados considera que os elementos não nulos estão presentes apenas nos blocos central, imediatamente anterior e superior, como mostra a Figura 6.4. Nesta figura, o bloco A é exatamente o mesmo utilizado no algoritmo SPIKE, e os blocos $\hat{C}_i$ e $\hat{B}_i$ são os blocos imediatamente anterior e posterior ao A, respectivamente, e de mesmas dimensões de A. Vale ressaltar que a diferença desses blocos para os blocos C_i e B_i do algoritmo clássico do SPIKE são as dimensões. C_i e B_i são de dimensão $\mathbf{k} \times \mathbf{k}$ no SPIKE e na operação SpMV, $\hat{C}_i$ e $\hat{B}_i$ possuem dimensão $n_i \times n_i$, onde n_i é o número de linhas encontradas pelo algoritmo de particionamento, em cada um dos processadores.

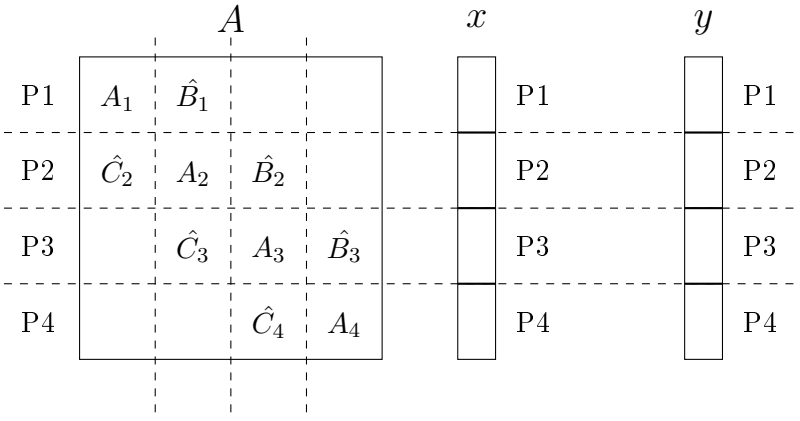


Figura 6.4: Decomposição 2D para o SpMV modificado para o SPIKE

A implementação utilizada neste trabalho considera uma divisão dos dados utilizando apenas memória distribuída com MPI, e portanto, não há leituras ou escritas concorrentes na memória relativa aos vetores x e y. Neste esquema, cada processador conhece um

bloco de linhas da matriz, o bloco correspondente do vetor de entrada x. O resultado da multiplicação é escrito no bloco correspondente do vetor saída y.

Além de toda a simplificação do SpMV bidimensional, durante o pré-processamento são calculadas as posições de coluna que ocupam o primeiro elemento não nulo dos blocos $\hat{B}^i$ e o último elemento não nulo dos blocos $\hat{C}_i$. Essa otimização reduz o tamanho correspondente do vetor x que precisa ser comunicado.

De maneira geral, o produto matriz-vetor paralelo $Ax = y$, considerando o particionamento do SPIKE, é definido nos p *ranks* MPI, como mostra a Equação 6.1.

$$\begin{array}{rcrcrcll} & & A_1x_1 & + & \hat{B}_1x_2 & = & y_1 & \text{(primeiro } \textit{rank}\text{)} \\ \hat{C}_ix_{i-1} & + & A_ix_i & + & \hat{B}_ix_{i+1} & = & y_i & \text{(}\textit{rank } p\text{)} \\ \hat{C}_px_{p-1} & + & A_px_p & + & & = & y_p & \text{(último } \textit{rank}\text{)} \end{array} \tag{6.1}$$

6.3 Pardiso

O pacote Pardiso (*Parallel Sparse Direct And Multi-Recursive Iterative Linear Solvers*) é um *software* de alto desempenho, robusto e eficiente usado para resolver sistemas lineares esparsos de grande porte em multiprocessadores de memória compartilhada e distribuída. Segundo Schenk and Gärtner [2014], desde sua primeira versão lançada em 2004, o Pardiso já foi utilizado por milhares de pesquisadores em universidades e laboratórios internacionais de computação científica.

Sua principal funcionalidade está na resolução de sistemas de equações cujas matrizes associadas podem ser: simétricas, não simétricas, estruturalmente simétricas, reais ou complexas, definidas positivas, indefinidas e hermitianas. Outras funcionalidades incluem a utilização da fatoração **LU**, com ou sem pivoteamento, e o uso de uma combinação automática de algoritmos baseados em métodos diretos e iterativos para acelerar a solução dos sistemas.

O Pardiso faz uso da BLAS (*Basic Linear Algebra Subprograms*), um conjunto de rotinas muito eficientes que implementam operações como produto escalar, vetorial, produto matriz-vetor e multiplicação de matrizes. A utilização do Pardiso requer a instalação de dois pacotes, BLAS e Lapack (*Linear Algebra PACKage*). Este último implementa algoritmos de álgebra linear, e também utiliza as rotinas presentes na BLAS. Adicionalmente, a fim de melhorar o desempenho de rotinas e operações matemáticas, podemos acoplar ao Pardiso à biblioteca MKL (*Math Kernel Library*) em arquiteturas Intel ou compatíveis. Dentre as vantagens do uso desta biblioteca estão o excelente desempenho de FFT (Transformada de *Fourier*), das operações de álgebra linear e funções da área de matemática e estatística [MKL, 2013].

Schenk and Gärtner [2014] relata que para problemas de grande porte, experimentos numéricos demonstraram que a escalabilidade do PARDISO é praticamente independente da arquitetura do multiprocessador de memória compartilhada ou distribuída. Além disso, foi observado um *speedup* de sete utilizando oito processadores.

As principais etapas do PARDISO são:

1. Reordenamento para reduzir o *fill-in* gerado na fatoração **LU** e Fatoração Simbólica que aloca a quantidade de memória necessária para essa fatoração.
2. Fatoração.
3. Substituição Retroativa e Refinamento Iterativo usado para recuperar e melhorar a qualidade das soluções numéricas, respectivamente.

Neste trabalho o *software* PARDISO será usado nas operações que envolvem métodos diretos no SPIKE, especificamente, durante a fatoração **LU**, através de rotinas que utilizam programação com memória compartilhada usando OpenMP.

Capítulo 7

Testes Computacionais

Os experimentos apresentados nesta seção foram todos realizados no *cluster* Altix-xe do Laboratório Nacional de Computação Científica (LNCC) localizado em Petrópolis, no Rio de Janeiro, exceto pelos testes presentes na subseção 7.4 onde analisamos o comportamento do precondicionador SPIKE em outros dois *clusters*. Sendo assim, falaremos por enquanto apenas das configurações do *cluster* Altix-xe (LNCC).

O *cluster* Altix-xe opera com 30 unidades de processamento, sendo 16 em funcionamento. Cada unidade possui 24GB de RAM, 8M de memória *cache* L2 e dois processadores Intel Xeon E5520 Quad Core, com frequência de *clock* de 2.27GHz, totalizando 8 *cores* por unidade. Neste *cluster*, o máximo de processadores permitido por usuário é 96.

Nossos códigos foram compilados com Intel versão 2015 *Update* 3 otimizados com a biblioteca *Math Kernel Library* MKL versão 11.2 *Update* 3 e MPI versão 5.0 *Update* 3. Todas as etapas do precondicionador SPIKE que envolvem cálculos usando métodos diretos foram resolvidos utilizando a biblioteca `pardiso500-INTEL1301-X86-64` do *software* PARDISO.

Para os experimentos realizados neste trabalho, aplicamos o método iterativo não-estacionário GMRES paralelo. As matrizes utilizadas ou foram obtidas do repositório de matrizes esparsas da Universidade da Flórida [Davis and Hu, 2011] ou geradas pelo autor pelo método de elementos finitos. Para as matrizes do repositório da Universidade da Flórida, o vetor de termos independentes foi obtido pela multiplicação da matriz pelo vetor trivial formado apenas por 1.0. Portanto, o método iterativo deve convergir para a solução 1.0 em todos esses sistemas.

A Tabela 7.1 apresenta as principais características do conjunto de todas as matrizes — de pequeno, médio e grande porte — presentes neste trabalho. As informações mostradas na tabela são o nome da matriz, dimensão (n), quantidade de elementos não nulos (nnz) e seu padrão de simetria.

Nome da matriz	Dimensão (n)	Não nulos (nnz)	Simétrica?
`rail_79841`	79.841	553.921	sim
`mario001`	38.434	204.912	sim
`atmosmodj`	1.270.432	8.814.880	não
`dw8192`	8.192	41.746	não
`G3_circuit`	1.585.478	7.660.826	sim
`parabolic_fem`	525.825	3.674.625	sim
`largebasis`	440.020	5.240.084	não
`CoupCons3D`	416.800	17.277.420	não
`Dubcova3`	146.689	3.636.643	sim
`nlpkkt120`	3.542.400	95.117.792	sim
`FEM_2D_3381977`	3.381.977	23.655.511	não
`FEM_3D_938586`	938.586	13.854.474	sim
`gemat_12`	4.929	33.044	não
`FEM_2D_1043474`	1.043.474	7.294.192	não

Tabela 7.1: Conjunto de matrizes testadas

Cada seção a seguir está relacionada com um teste específico. Primeiro, analisamos a influência do precondicionador SPIKE no conjunto das matrizes da Tabela 7.1. Em seguida, avaliamos a influência de cada estratégia combinatória separadamente e da melhor escolha possível dessas estratégias. Posteriormente, estudamos a influência do tamanho $\tilde{\mathbf{k}}$ das matrizes bloco de acoplamento e o efeito do uso de diferentes *clusters* com configurações distintas de *hardware* e *software*. Nos testes seguintes, examinamos o *speedup* e escalabilidade de duas aplicações onde as matrizes são derivadas de formulações de elementos finitos. Por fim, uma breve análise de desempenho dos algoritmos é apresentada.

7.1 Influência do Precondicionador

As análises, a seguir, visam mostrar a vantagem do uso do precondicionador SPIKE no método iterativo GMRES paralelo. Para todas as matrizes testadas nesta seção, foi utilizado memória distribuída com 8 MPI e apenas 1 *thread* OpenMP, o que significa que não foi usado memória compartilhada e, portanto, ainda podemos melhorar o tempo dos experimentos com precondicionador. Vale ressaltar que as estratégias combinatórias são feitas na etapa de pré-processamento de forma serial.

As informações desta seção estão organizadas como mostra a Figura 7.1:

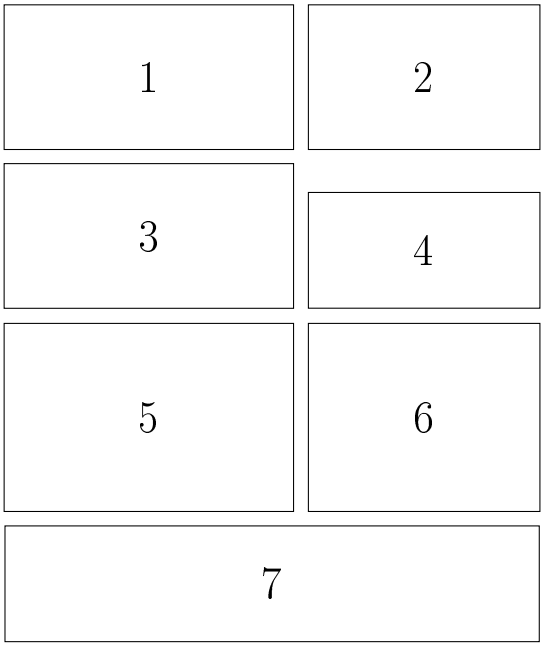


Figura 7.1: *Layout* das páginas: Influência do precondicionador

O bloco 1 mostra as propriedades da matriz analisada: dimensão, quantidade de não-nulos, estrutura da matriz e área de aplicação, e os parâmetros utilizados: tolerância, quantidade de vetores na base (*restart*), quantidade máxima de ciclos do algoritmo GMRES e tamanho $\tilde{\mathbf{k}}$ das matrizes bloco de acoplamento. O máximo de iterações é dado por (*restart*) $\times$ (quantidade máxima de ciclos). O símbolo †, presente nas tabelas, significa que o GMRES não convergiu após o número máximo de iterações.

O bloco 2 mostra as estratégias combinatórias utilizadas no processo de solução: *scaling*, *matching*, reordenamento Espectral (E) ou Espectral Valorado (EV), particionamento e problema quadrático da mochila (PQM). É importante destacar que, o *scaling* e o *matching* são realizados pelo mesmo algoritmo, o `MC64` (Seção 5.1).

O bloco 3 mostra uma TABELA com a quantidade de iterações e tempo de processamento do método iterativo GMRES na obtenção da solução, considerando execuções sem e com o precondicionador SPIKE. Neste último, a tabela mostra também a porcentagem de redução (quando houver) do tempo sem o precondicionador. No bloco 4 uma TABELA mostra o tempo de execução de cada estratégia combinatória utilizada.

O bloco 5 mostra uma FIGURA com a configuração da esparsidade da matriz original e configuração da esparsidade após a aplicação das estratégias combinatórias utilizadas e, no bloco 6, um gráfico de rosca representando a fração (porcentagem) do tempo das estratégias combinatórias utilizadas no tempo de solução total.

O bloco 7 faz uma análise de todos os dados apresentados nos blocos anteriores.

7.1.1 Matriz `rail_79841`

Propriedades e Parâmetros	
Dimensão	79.841
Não-nulos	553.921
Estrutura	Simétrica
Área de aplicação	Redução de Modelo
GMRES: Tolerância	10^{-8}
GMRES: *Restart* | Ciclos	100 | 1.000
$\tilde{\mathbf{k}}$ (blocos de acoplamento)	50

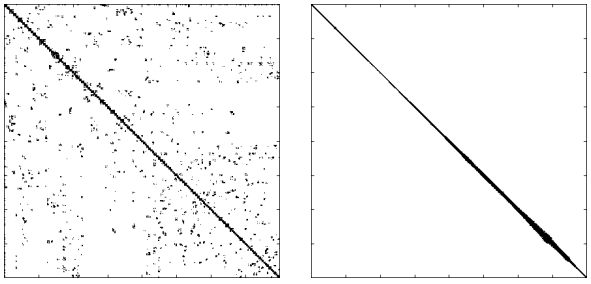
Sem precondicionador	
Iterações	43.201
Tempo de solução	91,373
Com precondicionador	
Iterações	313
Tempo de solução	2,685
% de redução do tempo	97%

Tabela 7.2: Iterações e tempo (seg) de solução do método iterativo GMRES

Estratégias combinatórias
Scaling
Reordenamento (EV)
Particionamento

Estratégias combinatórias	tempo
Scaling	0,032
Reordenamento	0,874
Particionamento	0,004
PQM	-

Tabela 7.3: Tempo (seg) das estratégias combinatórias



$lb = 79.811$ $lb = 1.718$

Figura 7.2: Configuração da esparsidade e largura de banda

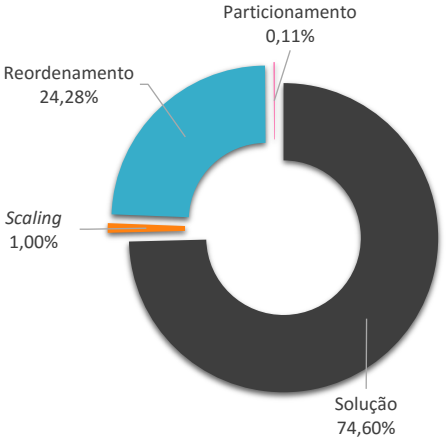


Figura 7.3: % do tempo das estratégias combinatórias no tempo de solução total do sistema precondicionado

Analisando os dados da Tabela 7.2 podemos perceber grande vantagem do uso do precondicionador SPIKE para essa matriz. A expressiva redução do número de iterações do GMRES proporcionou uma redução de 97% do tempo total de processamento. Na Figura 7.2 podemos observar grande redução da largura da banda da matriz após a aplicação das estratégias combinatórias, principalmente do reordenamento. Examinando a Tabela 7.3 e a Figura 7.3 vemos que o reordenamento Espectral Valorado representa quase um quarto (24,28%) do tempo total de solução. Esta porção é bastante significativa, o que reforça a ideia de implementar um algoritmo paralelo para o reordenamento.

7.1.2 Matriz `mario001`

Propriedades e Parâmetros	
Dimensão	38.434
Não-nulos	204.912
Estrutura	Simétrica
Área de aplicação	Problema 2D/3D
GMRES: Tolerância	10^{-8}
GMRES: *Restart* | Ciclos	100 | 1.000
$\tilde{\mathbf{k}}$ (blocos de acoplamento)	50

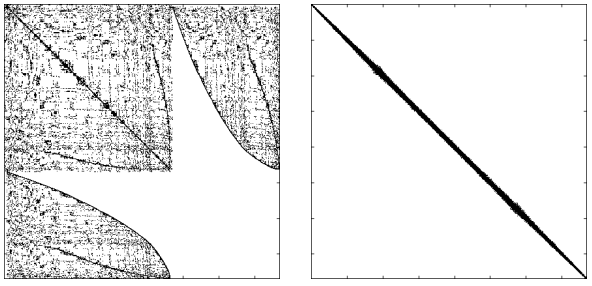
Sem precondicionador	
Iterações	†
Tempo de solução	†
Com precondicionador	
Iterações	199
Tempo de solução	0,926
% de redução do tempo	-

Tabela 7.4: Iterações e tempo (seg) de solução do método iterativo GMRES

Estratégias combinatórias
Matching e *Scaling*
Reordenamento (EV)
Particionamento
PQM

Estratégias combinatórias	tempo
Matching e *Scaling*	0,032
Reordenamento	0,364
Particionamento	0,002
PQM	0,062

Tabela 7.5: Tempo (seg) das estratégias combinatórias



$lb = 38.312$ $lb = 897$

Figura 7.4: Configuração da esparsidade e largura de banda

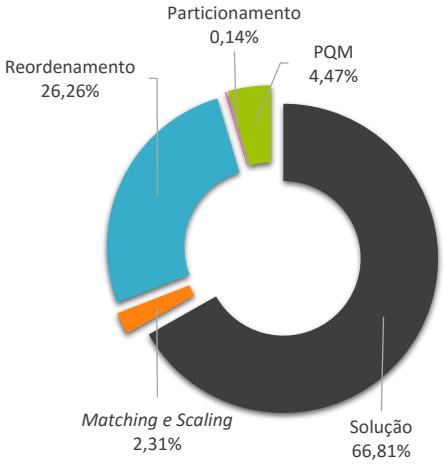


Figura 7.5: % do tempo das estratégias combinatórias no tempo de solução total do sistema precondicionado

A matriz em questão é de pequeno porte, porém com resultados bastante interessantes. Primeiramente, examinando a Tabela 7.4, percebemos que o GMRES não convergiu sem precondicionador, ao passo que com o precondicionador SPIKE sua convergência foi rápida. Em seguida, a Figura 7.4 mostra que a matriz original não possui diagonal principal bem definida, o que causaria inconsistências durante as etapas de solução do SPIKE que envolvem encontrar soluções via métodos diretos. Por esse motivo, a aplicação do *matching* foi essencial. Mais uma vez, o reordenamento Espectral Valorado representa boa parte do tempo total de solução, enquanto que o particionamento representa uma porção ínfima deste mesmo tempo (Figura 7.5).

7.1.3 Matriz `atmosmodj`

Propriedades e Parâmetros	
Dimensão	1.270.432
Não-nulos	8.814.880
Estrutura	Não simétrica
Área de aplicação	Dinâmica de Fluidos
GMRES: Tolerância	10^{-8}
GMRES: *Restart* | Ciclos	100 | 1.000
$\tilde{\mathbf{k}}$ (blocos de acoplamento)	50

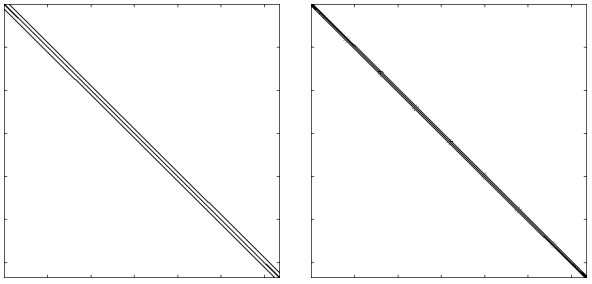
Sem precondicionador	
Iterações	664
Tempo de solução	19,106
Com precondicionador	
Iterações	82
Tempo de solução	237,703
% de redução do tempo	-1144%

Tabela 7.6: Iterações e tempo (seg) de solução do método iterativo GMRES

Estratégias combinatórias
Matching e *Scaling*
Reordenamento (E)
Particionamento
PQM

Estratégias combinatórias	tempo
Matching e *Scaling*	0,602
Reordenamento	10,428
Particionamento	0,046
PQM	2,902

Tabela 7.7: Tempo (seg) das estratégias combinatórias



$lb = 21.904$ $lb = 9.226$

Figura 7.6: Configuração da esparsidade e largura de banda

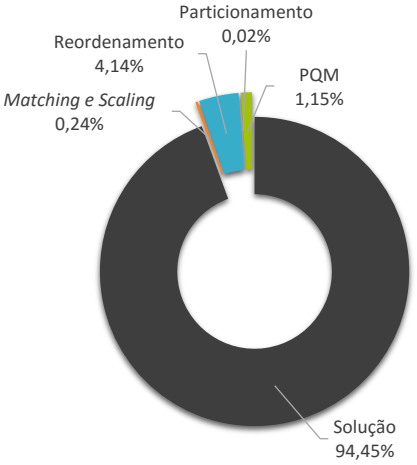


Figura 7.7: % do tempo das estratégias combinatórias no tempo de solução total do sistema precondicionado

A `atmosmodj` é uma matriz de grande porte que não apresentou vantagens ao ser resolvida com o precondicionador SPIKE. Na Tabela 7.6, a redução no número de iterações de 664 para 82 não foi suficiente para que a aplicação do precondicionador fosse vantajosa. Este fato ocorreu pois cada iteração do método GMRES com precondicionador SPIKE é muito mais custosa que o mesmo método sem precondicionador. Na Figura 7.6, observamos que o reordenamento Espectral não teve um efeito expressivo, visto que a largura de banda da matriz original já é bem estreita. Sendo assim, como o tempo de solução do GMRES foi muito alto, as estratégias combinatórias representaram uma pequena parte do tempo total, como mostra a Figura 7.7.

7.1.4 Matriz `dw8192`

Propriedades e Parâmetros	
Dimensão	8.192
Não-nulos	41.746
Estrutura	Não simétrica
Área de aplicação	Eletromagnetismo
GMRES: Tolerância	10^{-8}
GMRES: *Restart* | Ciclos	100 | 1.000
$\tilde{\mathbf{k}}$ (blocos de acoplamento)	50

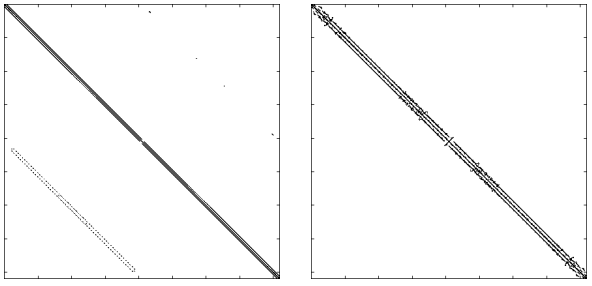
Sem precondicionador	
Iterações	†
Tempo de solução	†
Com precondicionador	
Iterações	39
Tempo de solução	0,090
% de redução do tempo	-

Tabela 7.8: Iterações e tempo (seg) de solução do método iterativo GMRES

Estratégias combinatórias
Reordenamento (EV)
Particionamento

Estratégias combinatórias	tempo
Matching e *Scaling*	-
Reordenamento	0,092
Particionamento	0,001
PQM	-

Tabela 7.9: Tempo (seg) das estratégias combinatórias



$lb = 4.160$ $lb = 255$

Figura 7.8: Configuração da esparsidade e largura de banda

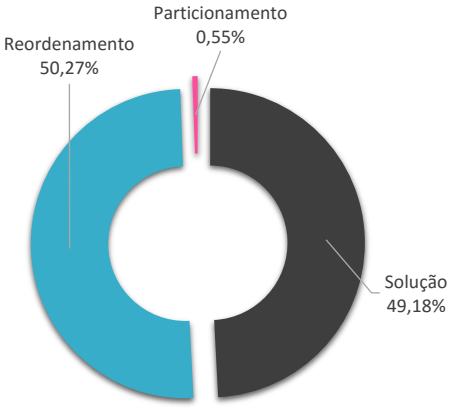


Figura 7.9: % do tempo das estratégias combinatórias no tempo de solução total do sistema precondicionado

Analisando os resultados obtidos para essa matriz, a Tabela 7.8 mostra que o método iterativo passou a convergir após a utilização do precondicionador SPIKE. Um fato interessante deste experimento é que o tempo gasto para realizar o reordenamento Espectral Valorado foi maior do que o tempo para o método GMRES resolver o sistema linear. Na Figura 7.9 isso fica bastante evidente quando 50,27% do tempo total foi gasto pelo reordenamento. Como o algoritmo SPIKE necessita que um conjunto de estratégias combinatórias sejam aplicadas na matriz, o ideal é que essas transformações sejam executadas em uma ordem de tempo muito menor que o restante da computação envolvida em resolver o sistema linear, fato que não ocorreu neste teste.

7.1.5 Matriz `G3_circuit`

Propriedades e Parâmetros	
Dimensão	1.585.478
Não-nulos	7.660.826
Estrutura	Simétrica
Área de aplicação	Simul. de Circuitos
GMRES: Tolerância	10^{-8}
GMRES: *Restart* | Ciclos	100 | 1.000
$\tilde{\mathbf{k}}$ (blocos de acoplamento)	50

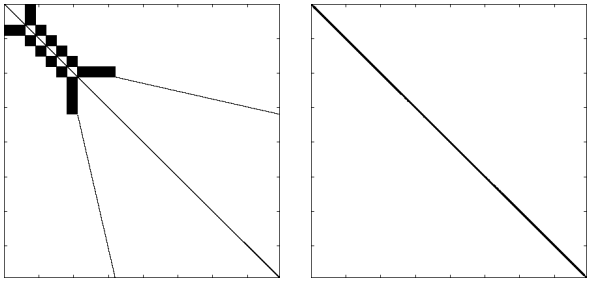
Sem precondicionador	
Iterações	4.104
Tempo de solução	170,782
Com precondicionador	
Iterações	256
Tempo de solução	83,612
% de redução do tempo	51%

Tabela 7.10: Iterações e tempo (seg) de solução do método iterativo GMRES

Estratégias combinatórias	
Matching e *Scaling*	
Reordenamento (E)	
Particionamento	
PQM	

Estratégias combinatórias	tempo
Matching e *Scaling*	0,613
Reordenamento	8,553
Particionamento	0,075
PQM	2,926

Tabela 7.11: Tempo (seg) das estratégias combinatórias



$lb = 947.128$ $lb = 9.067$

Figura 7.10: Configuração da esparsidade e largura de banda

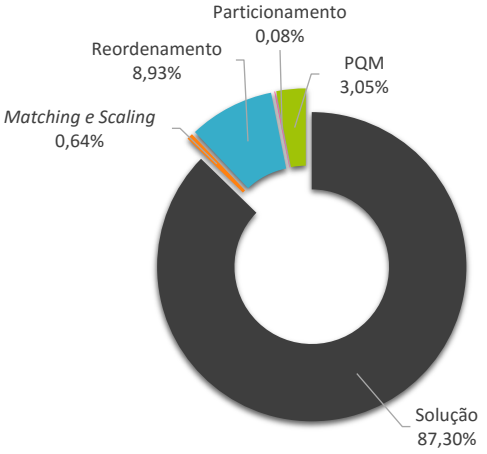


Figura 7.11: % do tempo das estratégias combinatórias no tempo de solução total do sistema precondicionado

Podemos ver na Tabela 7.10 que a matriz analisada possui uma convergência extremamente lenta e, neste caso, é recomendado o uso de um precondicionador. De fato, o precondicionador SPIKE reduziu muito o número de iterações do método GMRES e, consequentemente, diminuiu em 51% seu tempo de processamento. Todas as estratégias combinatórias foram aplicadas, sendo o reordenamento e o problema quadrático da mochila as que apresentaram tempo de execução mais significativos, como podemos observar na Tabela 7.11. A Figura 7.11 mostra que as estratégias combinatórias representaram, aproximadamente, 12,7% do tempo total de solução.

7.1.6 Matriz `parabolic_fem`

Propriedades e Parâmetros	
Dimensão	525.825
Não-nulos	3.674.625
Estrutura	Simétrica
Área de aplicação	Dinâmica de Fluidos
GMRES: Tolerância	10^{-6}
GMRES: *Restart* | Ciclos	100 | 1.000
$\tilde{\mathbf{k}}$ (blocos de acoplamento)	50

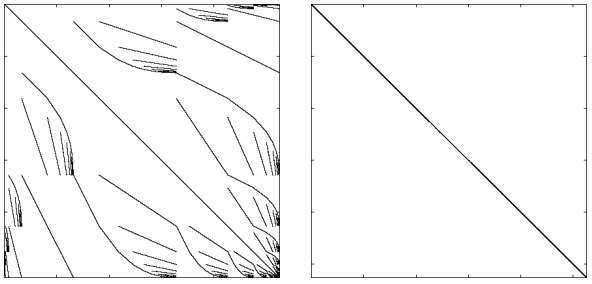
Sem precondicionador	
Iterações	10.891
Tempo de solução	149,401
Com precondicionador	
Iterações	245
Tempo de solução	26,25
% de redução do tempo	82%

Tabela 7.12: Iterações e tempo (seg) de solução do método iterativo GMRES

Estratégias combinatórias
Reordenamento (EV)
Particionamento

Estratégias combinatórias	tempo
Matching e *Scaling*	-
Reordenamento	5,820
Particionamento	0,019
PQM	-

Tabela 7.13: Tempo (seg) das estratégias combinatórias



$lb = 525.820$ $lb = 1.195$

Figura 7.12: Configuração da esparsidade e largura de banda

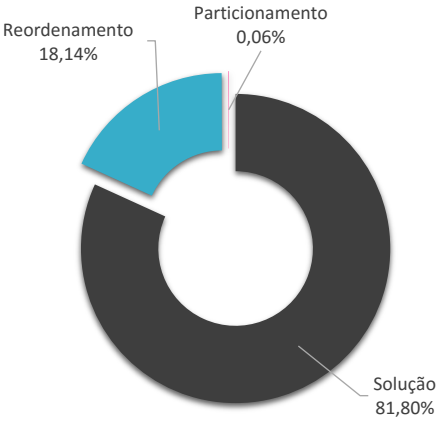


Figura 7.13: % do tempo das estratégias combinatórias no tempo de solução total do sistema precondicionado

A matriz `parabolic_fem`, obtida de um problema de dinâmica de fluidos, também apresentou convergência lenta no sistema sem precondicionador. Sendo assim, analisando a Tabela 7.12, o SPIKE demonstrou grande vantagem computacional diminuindo o número de iterações e reduzindo em 81% o tempo de processamento. As estratégias combinatórias utilizadas neste experimento foram apenas o particionamento e o reordenamento Espectral Valorado. Este último representou, aproximadamente, 18% do tempo total de processamento, como mostra o gráfico da Figura 7.13.

7.1.7 Matriz `largebasis`

Propriedades e Parâmetros	
Dimensão	440.020
Não-nulos	5.240.084
Estrutura	Não simétrica
Área de aplicação	Otimização
GMRES: Tolerância	10^{-8}
GMRES: *Restart* | Ciclos	100 | 1000
$\tilde{\mathbf{k}}$ (blocos de acoplamento)	50

Sem precondicionador	
Iterações	†
Tempo de solução	†
Com precondicionador	
Iterações	16
Tempo de solução	4,076
% de redução do tempo	-

Tabela 7.14: Iterações e tempo (seg) de solução do método iterativo GMRES

Estratégias combinatórias

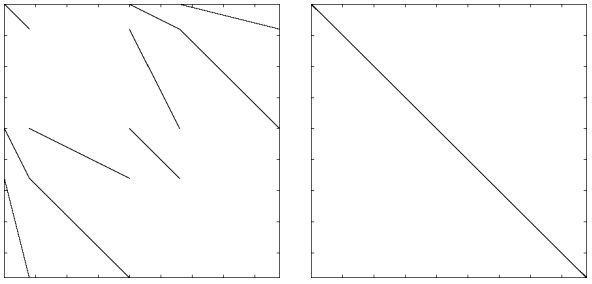
Matching e *Scaling*

Reordenamento (EV)

Particionamento

Estratégias combinatórias	tempo
Matching e *Scaling*	0,634
Reordenamento	12,532
Particionamento	0,024
PQM	-

Tabela 7.15: Tempo (seg) das estratégias combinatórias



$lb = 240.010$ $lb = 485$

Figura 7.14: Configuração da esparsidade e largura de banda

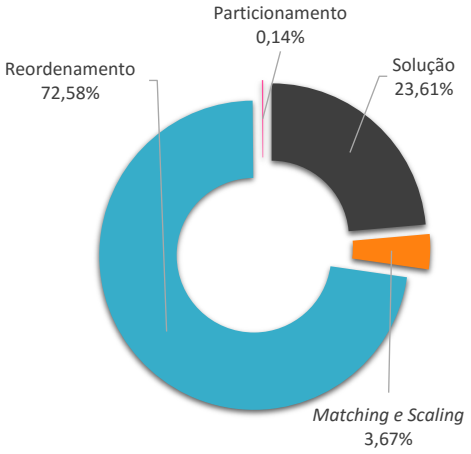


Figura 7.15: % do tempo das estratégias combinatórias no tempo de solução total do sistema precondicionado

Na tabela 7.14 observamos que, para essa matriz com aproximadamente 5 milhões de elementos não nulos, o GMRES não convergiu sem precondicionador. Por outro lado, a convergência foi muito rápida – apenas 16 iterações e 4,076 segundos – quando aplicamos o precondicionador SPIKE. Um tempo de solução tão pequeno evidenciou o tempo gasto com o reordenamento, que representou quase 73% do tempo total, como mostra o gráfico da Figura 7.15. Por fim, a configuração da esparsidade inicial, na Figura 7.14, mostra que grande parte da diagonal principal da matriz possui elementos nulos, o que torna imprescindível a utilização do *matching*.

7.1.8 Matriz `CoupCons3D`

Propriedades e Parâmetros	
Dimensão	416.800
Não-nulos	17.277.420
Estrutura	Não simétrica
Área de aplicação	Problema Estrutural
GMRES: Tolerância	10^{-8}
GMRES: *Restart* | Ciclos	100 | 1000
$\tilde{\mathbf{k}}$ (blocos de acoplamento)	50

Sem precondicionador	
Iterações	89.799
Tempo de solução	2.964,340
Com precondicionador	
Iterações	21
Tempo de solução	38,198
% de redução do tempo	99%

Tabela 7.16: Iterações e tempo (seg) de solução do método iterativo GMRES

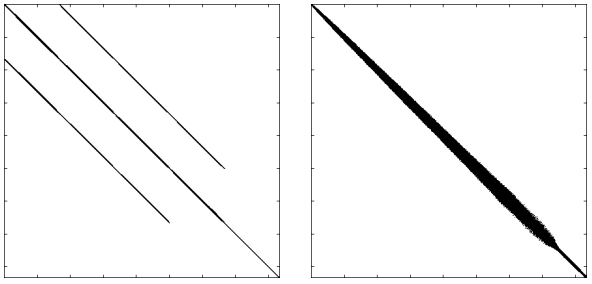
Estratégias combinatórias

Reordenamento (E)

Particionamento

Estratégias combinatórias	tempo
Matching e *Scaling*	-
Reordenamento	23,961
Particionamento	0,021
PQM	-

Tabela 7.17: Tempo (seg) das estratégias combinatórias



$lb = 83.363$ $lb = 18.039$

Figura 7.16: Configuração da esparsidade e largura de banda

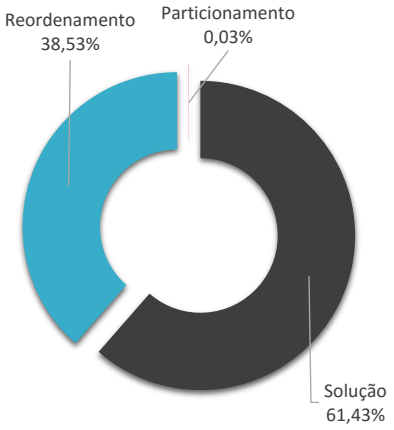


Figura 7.17: % do tempo das estratégias combinatórias no tempo de solução total do sistema precondicionado

A matriz `CoupCons3D` possui grande quantidade de não nulos, apesar de uma dimensão mediana, o que faz dela uma matriz de grande porte. Na Tabela 7.16 vemos que o tempo de solução do GMRES sem precondicionador foi extremamente alto em comparação ao tempo de solução com o precondicionador SPIKE. Com isso, foi possível alcançar uma excelente redução de 99% do tempo de processamento. O número de iterações passou de 89.799 para 21, uma queda bastante expressiva. Mais uma vez, o reordenamento Espectral foi responsável por grande parte do tempo total de solução, o que reforça a implementação de um algoritmo paralelo para essa estratégia.

7.1.9 Matriz `Dubcova3`

Propriedades e Parâmetros	
Dimensão	146.689
Não-nulos	3.636.643
Estrutura	Simétrica
Área de aplicação	Problema 2D/3D
GMRES: Tolerância	10^{-12}
GMRES: *Restart* | Ciclos	100 | 1000
$\tilde{\mathbf{k}}$ (blocos de acoplamento)	2

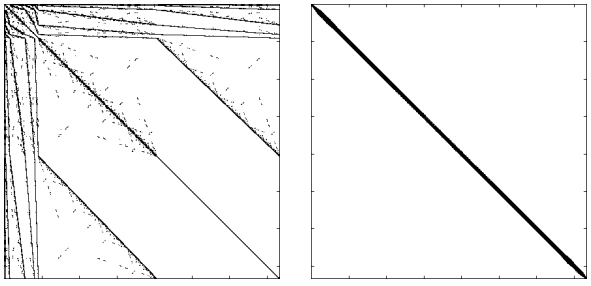
Sem precondicionador	
Iterações	585
Tempo de solução	3,692
Com precondicionador	
Iterações	99
Tempo de solução	3,055
% de redução do tempo	17%

Tabela 7.18: Iterações e tempo (seg) de solução do método iterativo GMRES

Estratégias combinatórias
Scaling
Reordenamento (EV)
Particionamento

Estratégias combinatórias	tempo
Scaling	0,172
Reordenamento	2,899
Particionamento	0,008
PQM	-

Tabela 7.19: Tempo (seg) das estratégias combinatórias



$lb = 146.356$ $lb = 1.803$

Figura 7.18: Configuração da esparsidade e largura de banda

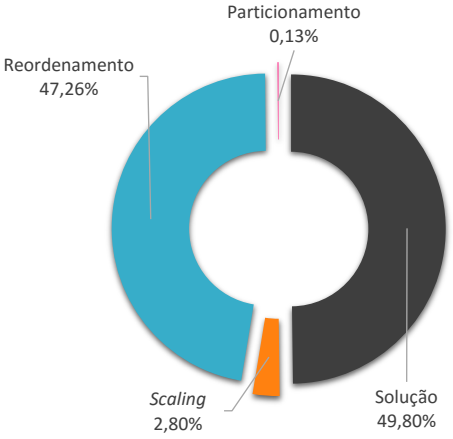


Figura 7.19: % do tempo das estratégias combinatórias no tempo de solução total do sistema precondicionado

A Tabela 7.18 mostra que não houve uma queda muito expressiva na quantidade de iterações e nem no tempo de processamento dessa matriz. A pequena redução de 17% só foi alcançada pois houve uma boa escolha de parâmetros, como a redução do tamanho $\tilde{\mathbf{k}}$ das matrizes de bloco acoplamento de 50 para 2 e o aumento da tolerância do GMRES para 10^{-12}. Neste teste, qualquer outra configuração de parâmetros não é vantajosa para o precondicionador SPIKE. Como já foi dito, as iterações do SPIKE são mais custosas e, portanto, é preciso que a matriz seja muito mal condicionada para obtermos alguma vantagem. A largura de banda dessa matriz foi bastante reduzida, como mostra a Figura 7.18, e o reordenamento levou quase metade do tempo total de solução (Figura 7.19).

7.1.10 Matriz `nlpkkt120`

Propriedades e Parâmetros	
Dimensão	3.542.400
Não-nulos	95.117.792
Estrutura	Simétrica
Área de aplicação	Otimização
GMRES: Tolerância	10^{-8}
GMRES: *Restart* | Ciclos	100 | 100
$\tilde{\mathbf{k}}$ (blocos de acoplamento)	5

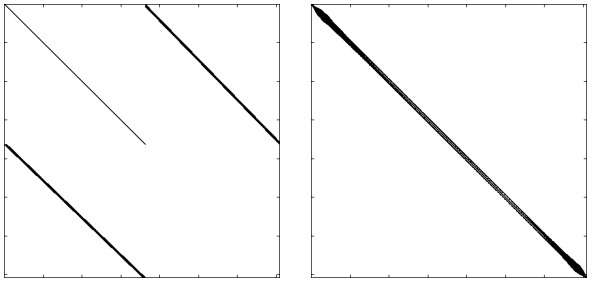
Sem precondicionador	
Iterações	†
Tempo de solução	†
Com precondicionador	
Iterações	102
Tempo de solução	4865,998
% de redução do tempo	-

Tabela 7.20: Iterações e tempo (seg) de solução do método iterativo GMRES

Estratégias combinatórias
Matching e *Scaling*
Reordenamento (EV)
Particionamento
PQM

Estratégias combinatórias	tempo
Matching e *Scaling*	5,444
Reordenamento	116,388
Particionamento	0,141
PQM	26,001

Tabela 7.21: Tempo (seg) das estratégias combinatórias



$lb = 1.799.879$ $lb = 64.214$

Figura 7.20: Configuração da esparsidade e largura de banda

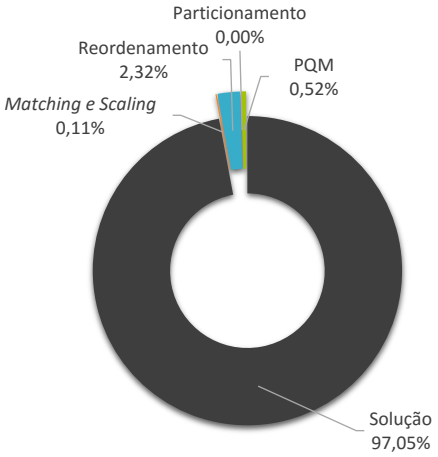


Figura 7.21: % do tempo das estratégias combinatórias no tempo de solução total do sistema precondicionado

Dentre as matrizes desta seção, a matriz `nlpkkt120` é a que possui a maior quantidade de elementos não nulos. Com isso, cada um dos 8 processadores precisou lidar com cerca de 12 milhões de elementos não nulos. Como podemos perceber na Figura 7.20, o *matching* é necessário para preencher toda a diagonal principal de elementos não nulos. A Tabela 7.20 mostra que, sem precondicionador, o GMRES atingiu o máximo de iterações sem convergir e com precondicionador SPIKE, o método foi capaz de convergir com 102 iterações. O elevado tempo de solução do sistema (97%) fez o tempo das estratégias combinatórias pouco significativas no tempo total, como mostra a Figura 7.21.

7.1.11 Matriz `FEM_2D_3381977`

Propriedades e Parâmetros	
Dimensão	3.381.977
Não-nulos	23.655.511
Estrutura	Não simétrica
Área de aplicação	Elementos Finitos
GMRES: Tolerância	10^{-8}
GMRES: *Restart* | Ciclos	100 | 100
$\tilde{\mathbf{k}}$ (blocos de acoplamento)	10

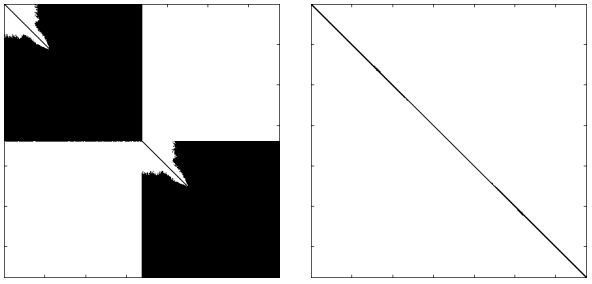
Sem precondicionador	
Iterações	8.346
Tempo de solução	1247,411
Com precondicionador	
Iterações	50
Tempo de solução	67,640
% de redução do tempo	95%

Tabela 7.22: Iterações e tempo (seg) de solução do método iterativo GMRES

Estratégias combinatórias
Scaling
Reordenamento (EV)
Particionamento

Estratégias combinatórias	tempo
Scaling	7,791
Reordenamento	63,025
Particionamento	0,166
PQM	-

Tabela 7.23: Tempo (seg) das estratégias combinatórias



$lb = 3.379.552$ $lb = 13.549$

Figura 7.22: Configuração da esparsidade e largura de banda

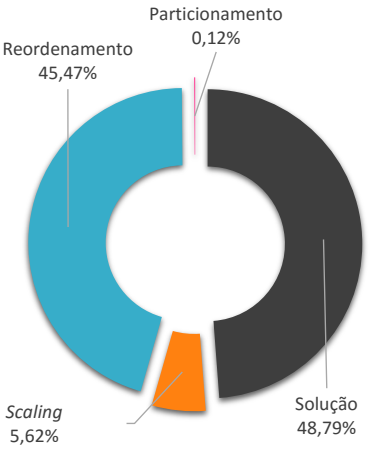


Figura 7.23: % do tempo das estratégias combinatórias no tempo de solução total do sistema precondicionado

A matriz `FEM_2D_3381977` foi gerada pelo autor pelo método de elementos finitos, a partir da equação de advecção difusão estacionária bidimensional. Detalhes sobre esse problema serão descritos na subseção 7.5.1. Neste teste, o algoritmo SPIKE alcançou uma excelente redução de 95% do tempo de processamento, em consequência da redução expressiva do número de iterações de 8.346 para 50 (ver Tabela 7.23). O reordenamento Espectral Valorado, mais uma vez, representou grande parte do tempo total de solução, como mostra o gráfico da Figura 7.23. Também é interessante observar a grande redução da largura de banda da matriz na Figura 7.23, que passou de uma estrutura de blocos para uma composição bastante estreita, próximo a diagonal principal.

7.1.12 Matriz `FEM_3D_938586`

Propriedades e Parâmetros	
Dimensão	938.586
Não-nulos	13.854.474
Estrutura	Simétrica
Área de aplicação	Elementos Finitos
GMRES: Tolerância	10^{-8}
GMRES: *Restart* | Ciclos	100 | 100
$\tilde{\mathbf{k}}$ (blocos de acoplamento)	50

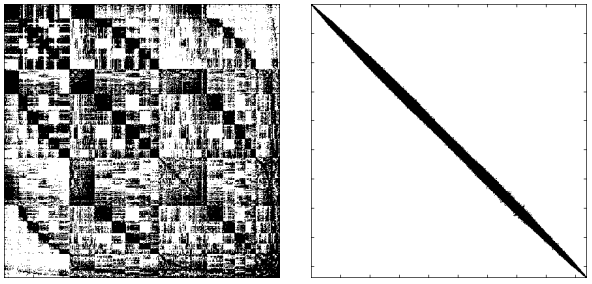
Sem precondicionador	
Iterações	6.334
Tempo de solução	357,895
Com precondicionador	
Iterações	94
Tempo de solução	237,397
% de redução do tempo	34%

Tabela 7.24: Iterações e tempo (seg) de solução do método iterativo GMRES

Estratégias combinatórias
Scaling
Reordenamento (EV)
Particionamento

Estratégias combinatórias	tempo
Scaling	3,186
Reordenamento	21,790
Particionamento	0,061
PQM	-

Tabela 7.25: Tempo (seg) das estratégias combinatórias



$lb = 937.528$ $lb = 35.204$

Figura 7.24: Configuração da esparsidade e largura de banda

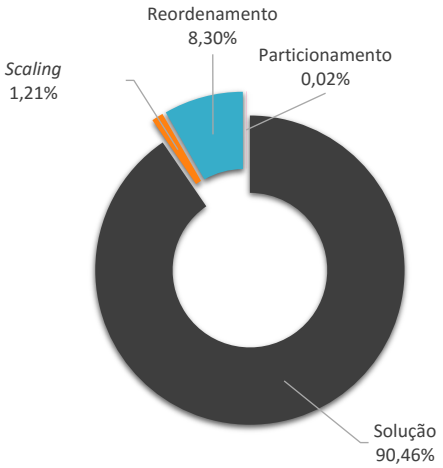


Figura 7.25: % do tempo das estratégias combinatórias no tempo de solução total do sistema precondicionado

A matriz `FEM_3D_938586` também foi gerada pelo autor, oriunda de um problema tridimensional de transferência de calor, cujo domínio foi discretizado pelo método dos elementos finitos. Detalhes deste problema podem ser vistos na subseção 7.5.2. Apesar da grande redução do número de iterações, o tempo de execução não reduziu na mesma proporção, como mostra a Tabela 7.25. Isto se deve ao fato de que cada iteração com o precondicionador SPIKE foi bem custosa, levando a uma redução de apenas 34% no tempo final de execução. Observando a Figura 7.25, o alto tempo de solução fez com que o tempo das estratégias combinatórias representassem uma pequena porção do tempo total de solução do GMRES, sendo o reordenamento o mais significativo.

7.2 Influência das Estratégias Combinatórias

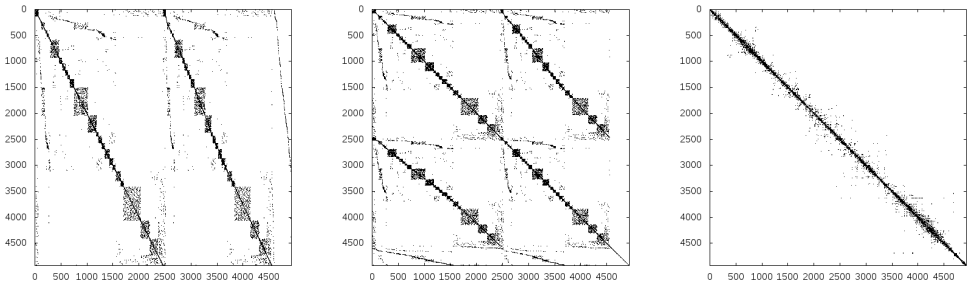
Nessa seção, analisamos o comportamento do método GMRES com o precondicionador SPIKE quando fixamos cada uma das estratégias combinatórias separadamente. Isto é, uma vez estabelecida uma configuração de estratégias combinatórias, iremos ligar e desligar uma delas para avaliar sua influência no número de iterações e tempos de execução do GMRES. Todos os testes apresentados a seguir foram executados no *cluster* Altix-xe, variando a quantidade p de processadores distribuídos. Não foi usado memória compartilhada. Mesmo as matrizes que já foram apresentadas na seção anterior, terão suas propriedades e parâmetros escolhidos exibidos novamente visto que, nos experimentos desta seção, alguns desses parâmetros foram modificados.

7.2.1 *Matching*

O *matching* é usado para mover os elementos mais significativos, em módulo, para diagonal principal. Esta técnica é fundamental para matrizes que não possuem elementos na diagonal principal, pois além da dificuldade de realizar a fatoração **LU**, outras estratégias também não apresentam um bom comportamento. Analisamos a matriz `gemat12` que, em sua configuração original, não possui a diagonal principal totalmente preenchida e, por isso, não converge. Além disso, o reordenamento Espectral encontra dificuldades em reduzir sua largura de banda.

Propriedades e Parâmetros	
Dimensão	4.929
Não-nulos	33.044
Estrutura	Não simétrica
Área de aplicação	Redes Elétricas
GMRES: Tolerância	10^{-8}
GMRES: *Restart* | Ciclos	50 | 1.000
$\tilde{\mathbf{k}}$ (blocos de acoplamento)	50

A Figura 7.26 mostra a matriz `gemat12` em três estágios: (a) configuração de esparsidade original, (b) após a aplicação do *matching* e (c) seguido o reordenamento Espectral.



(a) (b) (c)

Figura 7.26: Influência do *matching* - Configuração da esparsidade

7.2.2 *Scaling*

A Tabela 7.26 mostra a influência na construção do precondicionador SPIKE quando aplicamos ou não o *scaling* na matriz `rail_79841`. As mudanças de parâmetros com relação a mesma matriz apresentada na seção 7.1.1 são o número de vetores *restart* do GMRES e mudança do algoritmo de reordenamento. O reordenamento Espectral Valorado não apresentou bom comportamento quando não usamos o *scaling* e por isso, neste experimento, foi substituído pelo Espectral padrão. Este fato mostra a importância da aplicação do *scaling* em determinadas matrizes, como esta em questão.

Propriedades e Parâmetros	
Dimensão	79.841
Não-nulos	553.921
Estrutura	Simétrica
Área de aplicação	Redução de Modelo
GMRES: Tolerância	10^{-8}
GMRES: *Restart* | Ciclos	50 | 1.000
$\tilde{\mathbf{k}}$ (blocos de acoplamento)	50

Estratégias combinatórias

- *Scaling*
- Reordenamento (E)
- Particionamento

sem *scaling*	p=2	p=4	p=8	p=16
Iterações	150	549	1.644	4.640
Tempo (seg)	5,60	9,34	15,37	16,16
com *scaling*	p=2	p=4	p=8	p=16
Iterações	98	329	791	2.296
Tempo (seg)	3,92	5,77	7,51	8,08

Tabela 7.26: Influência do *scaling*: Iterações e tempo de execução do GMRES

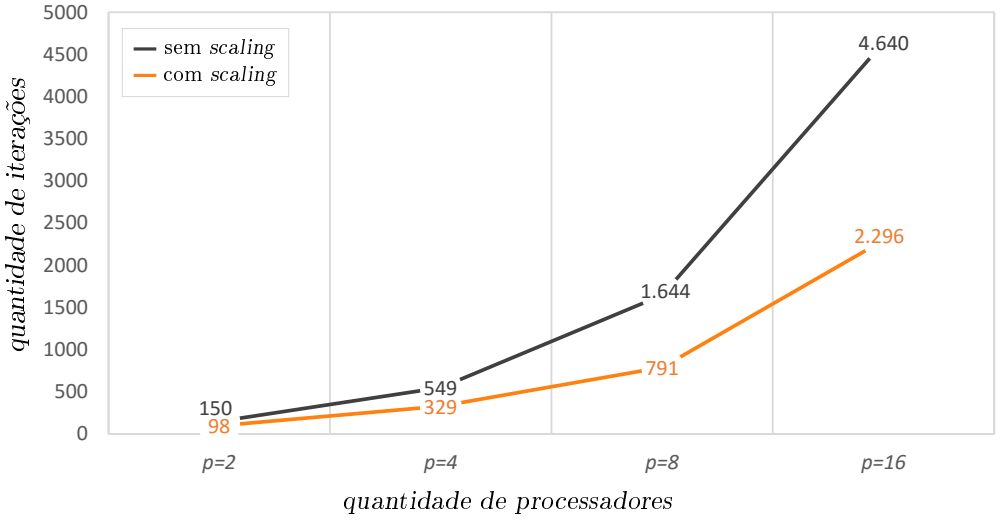


Figura 7.27: Influência do *scaling*: Iterações do GMRES

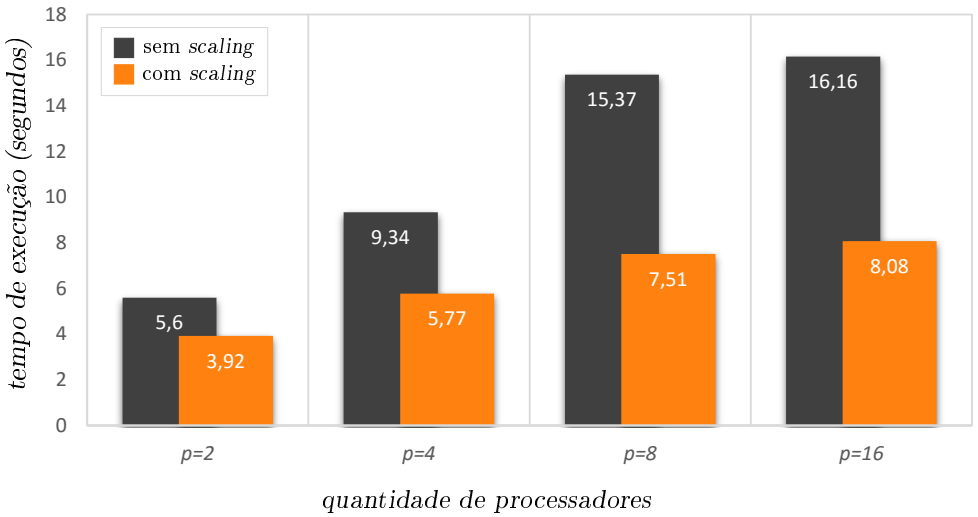


Figura 7.28: Influência do *scaling*: Tempo de execução (seg) do GMRES

Os dados da Tabela 7.26 mostram a expressiva redução do número de iterações e do tempo de processamento quando usamos o *scaling*. Na Figura 7.27 podemos observar o comportamento das iterações do GMRES à medida que aumentamos a quantidade de processadores distribuídos. Nessa figura, conforme a quantidade de processadores aumentou, maior foi a redução do número de iterações do GMRES quando usamos o *scaling*. O melhor resultado ocorreu com 8 processadores, onde foi possível reduzir em mais que 51% o número de iterações.

A redução do número de iterações impacta diretamente no tempo de processamento, como podemos observar na Figura 7.28 que exibe os tempos de execução do GMRES sem e com o *scaling*. Com relação ao tempo de processamento, houve redução para todas as configurações de processadores distribuídos testadas, sendo que a mais significativa, representando 51%, ocorre quando utilizamos 8 processadores.

Vale destacar que o precondicionador SPIKE, neste teste, não apresentou um bom *speedup*. Isso ocorre porque à medida que a quantidade de processadores aumenta, o número de iterações do método GMRES tende a crescer. Sendo assim, como essa matriz é pequena com relação ao poder de processamento do *cluster*, o ganho alcançado pela divisão de trabalho computacional não é suficiente para superar os gastos de realizar um maior número de iterações somado aos gastos de comunicação entre processadores.

Como observado, o *scaling* pode ser uma estratégia combinatória de grande impacto no precondicionador SPIKE, reduzindo o número de iterações necessárias para convergência do GMRES e, consequentemente, seu tempo de execução.

7.2.3 Reordenamento

Enquanto o reordenamento Espectral não se preocupa com a magnitude dos elementos, o reordenamento Espectral Valorado tem por objetivo mover os elementos mais significativos para próximo da diagonal principal, podendo exercer bastante influência na convergência do método iterativo. A Tabela 7.27 detalha o impacto desse reordenamento aplicado à matriz `dw8192`. A única mudança de parâmetro com relação a mesma matriz apresentada na seção 7.1.4 foi o número de vetores *restart* do GMRES que, neste teste, foi escolhido um valor alto para que o precondicionador, com ambos os reordenamentos, fosse capaz de convergir.

Propriedades e Parâmetros	
Dimensão	8.192
Não-nulos	41.746
Estrutura	Não simétrica
Área de aplicação	Eletromagnetismo
GMRES: Tolerância	10^{-8}
GMRES: *Restart* | Ciclos	2.000 | 1.000
$\tilde{\mathbf{k}}$ (blocos de acoplamento)	50

Estratégias combinatórias
Reordenamento (E | EV)
Particionamento

Espectral	p=2	p=4	p=8	p=16
Iterações	196	614	1.165	1.992
Tempo (seg)	0,83	2,28	6,04	18,81
Espectral Valorado	p=2	p=4	p=8	p=16
Iterações	14	27	42	88
Tempo (seg)	0,15	0,11	0,09	0,10

Tabela 7.27: Influência do reordenamento: Iterações e tempo de execução do GMRES

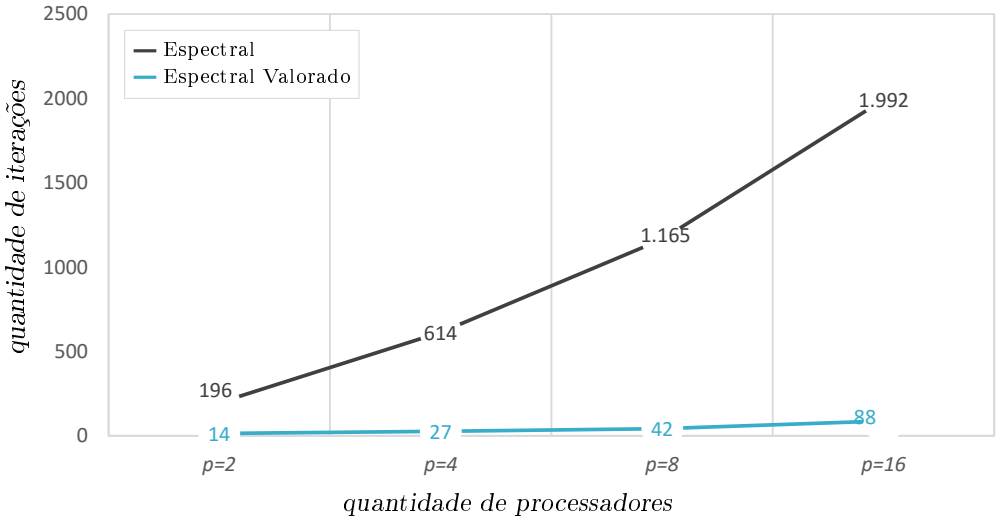


Figura 7.29: Influência do reordenamento: Iterações do método GMRES

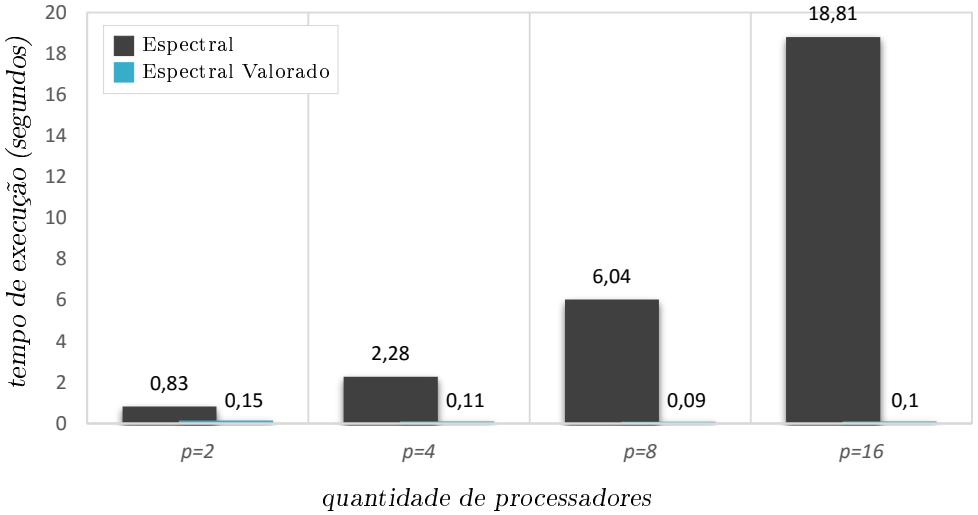


Figura 7.30: Influência do reordenamento: Tempo de execução (seg) do GMRES

A Tabela 7.27 apresenta o número de iterações e tempo de execução do GMRES quando aplicamos o reordenamento Espectral ou o Espectral Valorado. Neste caso, como o reordenamento é sempre necessário quando não temos uma matriz com estrutura de banda, não faria sentido não usar a estratégia como fizemos no teste da subseção anterior. Portanto, nesta seção, iremos comparar a influência do reordenamento Espectral Valorado com relação ao Espectral padrão.

Analisando a Figura 7.29 observamos um crescimento acentuado das iterações do GMRES quando utilizamos o reordenamento Espectral, conforme aumentamos a quantidade de processadores. Em contrapartida, as iterações crescem suavemente ao longo dos processadores quando usamos o reordenamento Espectral Valorado. Além disso, em cada configuração de processadores distribuídos, as iterações do Espectral Valorado são muito menores se comparadas às do Espectral padrão. Destaque para uma redução de, aproximadamente, 96% do número de iterações em 16 processadores.

O gráfico da Figura 7.30 mostra uma grande redução do tempo de processamento ao utilizarmos o Espectral Valorado. Enquanto os tempos do Espectral padrão cresceram exponencialmente, os do Espectral Valorado reduziram lentamente ao longo dos processadores de forma que foi possível obter um pequeno *speedup*. A redução mais expressiva, de mais de 99%, ocorreu quando utilizamos 16 processadores distribuídos.

Em síntese, o reordenamento pode causar grande influência no precondicionador SPIKE. Neste caso, mover os elementos mais significativos para mais próximo da diagonal principal teve um impacto positivo na solução do método GMRES.

7.2.4 Problema Quadrático da Mochila (PQM)

A ação de mover os elementos mais significativos para dentro dos blocos de acoplamento B_i e C_i pode ser muito benéfica. A Tabela 7.28 mostra o efeito dessa estratégia combinatória na matriz `mario001`, mostrando a quantidade de iterações, tempo do GMRES e a quantidade de elementos movidos para dentro das matrizes bloco de acoplamento. Os parâmetros modificados com relação a mesma matriz apresentada na seção 7.1.2 são o número de vetores *restart* do GMRES e tamanho $\tilde{\mathbf{k}}$ desses blocos de acoplamento.

Propriedades e Parâmetros	
Dimensão	38.434
Não-nulos	204.912
Estrutura	Simétrica
Área de aplicação	Problema 2D/3D
GMRES: Tolerância	10^{-8}
GMRES: *Restart* | Ciclos	50 | 1.000
$\tilde{\mathbf{k}}$ (blocos de acoplamento)	100

Estratégias combinatórias
Matching e *Scaling*
Reordenamento (EV)
Particionamento
PQM

sem PQM	p=2	p=4	p=8	p=16
Iterações	44	144	369	1.966
Tempo (seg)	1,233	1,599	1,757	4,552
com PQM	p=2	p=4	p=8	p=16
Iterações	48	100	140	248
Tempo (seg)	1,467	1,248	0,838	0,806
Elementos movidos	114	399	765	1.522

Tabela 7.28: Influência do PQM: Iterações e tempo de execução do GMRES

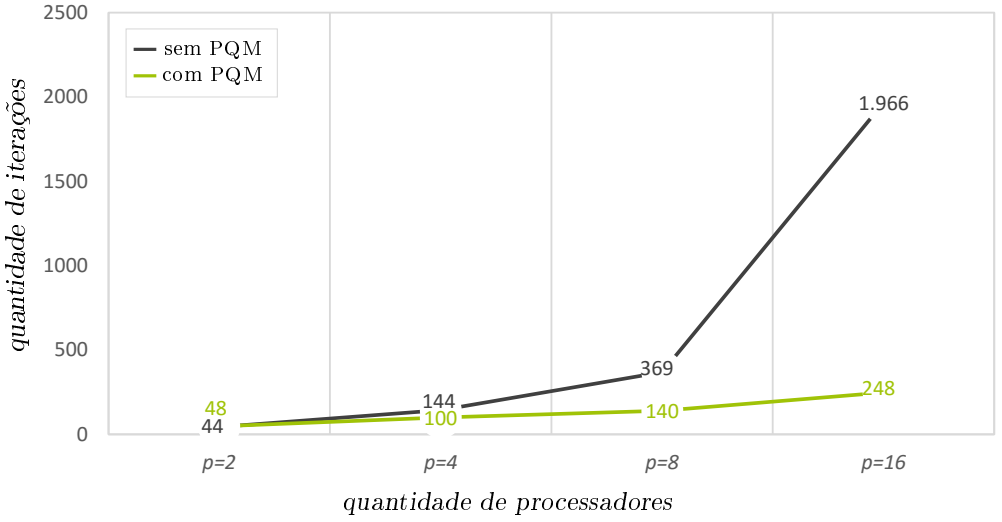


Figura 7.31: Influência do PQM: Iterações do método GMRES

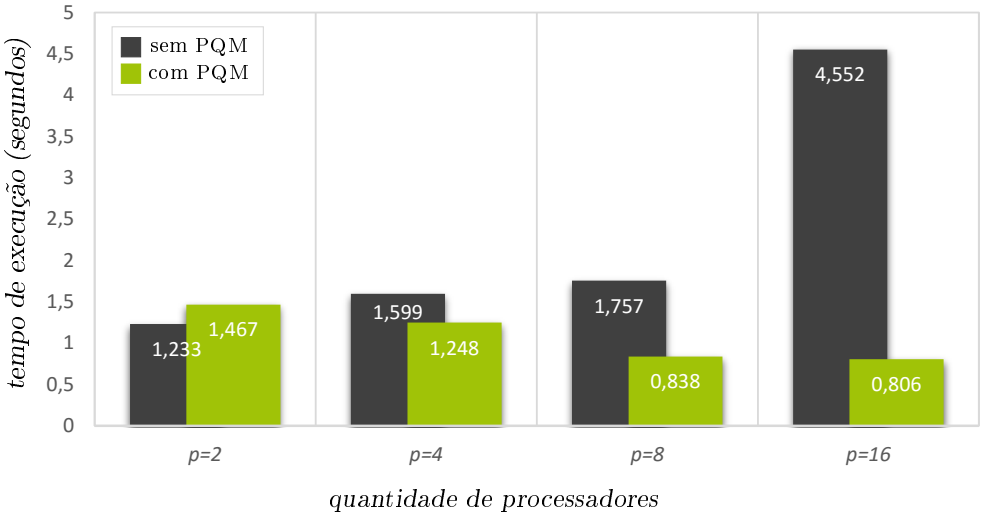


Figura 7.32: Influência do PQM: Tempo de execução (seg) do GMRES

A Tabela 7.28 nos mostra, além das iterações e tempo de processamento do GMRES, a quantidade de elementos movidos para dentro das matrizes bloco de acoplamento B_i e C_i quando utilizamos a estratégia combinatória modelada como problema quadrático da mochila. Podemos notar que quando aumentamos o número de processadores, mais elementos o algoritmo é capaz de mover para dentro dos blocos de acoplamento.

Ao observamos a Figura 7.31 vemos que, conforme aumentamos a quantidade de processadores, o número de iterações também aumenta quando não utilizamos o PQM, exceto para dois processadores. O aumento de 369 para 1.966 é bastante expressivo quando o número de processadores passa de 8 para 16. Por outro lado, quando utilizamos o PQM, temos um discreto crescimento no número de iterações do GMRES.

Com relação ao tempo de processamento, na Figura 7.32, notamos que, com dois processadores, utilizar o PQM não foi vantajoso dado que o tempo de solução aumentou. Entretanto, para as demais configurações de processadores, a estratégia combinatória em questão foi eficaz, com ênfase para uma redução de 82% do tempo total de processamento com 16 processadores.

Sendo assim, o problema quadrático da mochila também pode exercer grande influência no precondicionador SPIKE. Porém, nos testes realizados na seção anterior, percebemos que essa estratégia combinatória é a que apresenta resultados menos satisfatórios, isto é, frequentemente o número de iterações e tempo de processamento aumentam quando a utilizamos. Este fato pode estar relacionado a detalhes de implementação em consequência da limitada bibliografia existente sobre o algoritmo *DeMin*, o que sugere mais estudos com relação a essa estratégia combinatória.

7.2.5 Melhor escolha de Estratégias Combinatórias

O objetivo do teste a seguir é mostrar a influência das estratégias combinatórias no precondicionador SPIKE. Diferente do teste anterior onde avaliamos cada estratégia separadamente, aqui comparamos a escolha mais simples de estratégias combinatórias, i.e., as estratégias indispensáveis, com a melhor escolha dessas estratégias.

Assim como a matriz da Seção 7.1.11, a matriz utilizada foi gerada pelo autor e obtida pelo mesmo problema de elementos finitos, porém de uma malha menor. Foi denotada como `FEM_2D_1043474` e suas propriedades e parâmetros estão descritos abaixo.

Propriedades e Parâmetros	
Dimensão	1.043.474
Não-nulos	7.294.192
Estrutura	Não simétrica
Área de aplicação	Elementos Finitos
GMRES: Tolerância	10^{-8}
GMRES: *Restart* | Ciclos	50 | 1000
$\tilde{\mathbf{k}}$ (blocos de acoplamento)	50

Nas tabelas abaixo variamos o número de nós MPI e de *threads* OpenMP e, dessa forma, a quantidade total de processadores é **nós distribuídos** × ***threads***. As tabelas mostram iterações e tempo, em segundos, de uma execução do GMRES com o SPIKE. A Tabela 7.29 mostra os dados utilizando somente o Espectral, já que o reordenamento é indispensável. Na Tabela 7.30 escolhemos a melhor configuração de estratégias possível que, neste caso, foi o Espectral Valorado com o *scaling*. O melhor tempo é destacado nas tabelas.

		OpenMP		
MPI	Iterações	1	2	4
2	26	45,59	34,00	26,09
4	42	31,22	22,80	26,61
8	68	17,94	13,37	13,20
16	80	9,11	6,80	6,61

Tabela 7.29: Estratégias: Espectral padrão

		OpenMP		
MPI	Iterações	1	2	4
2	20	38,88	25,73	19,85
4	33	28,46	21,57	20,67
8	39	13,19	9,32	12,84
16	56	7,68	5,36	7,35

Tabela 7.30: Estratégias: Espectral Valorado + *scaling*

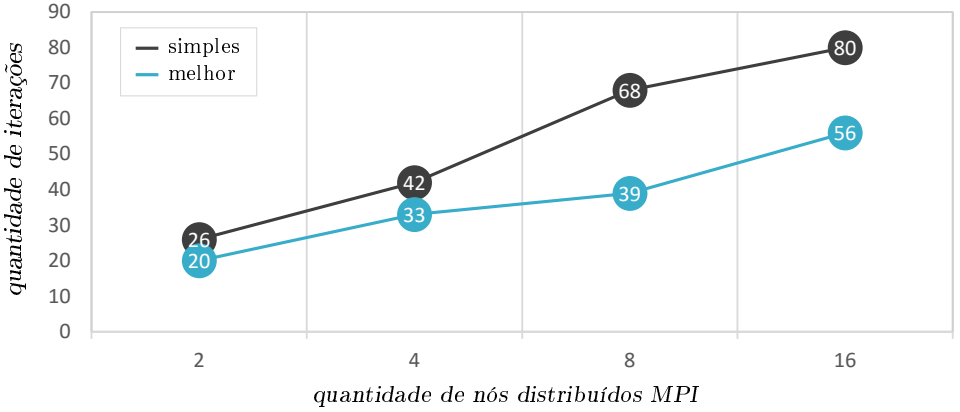


Figura 7.33: Escolha mais simples × melhor escolha de estratégias: Iterações

A Figura 7.33 mostra o número de iterações do método GMRES para 2, 4, 8 e 16 nós MPI, comparando a escolha mais simples com a melhor escolha de estratégias combinatórias. Como podemos observar nas tabelas, o número de iterações não muda quando aumentamos a quantidade de *threads* OpenMP. Isso acontece porque a variação da quantidade de *threads* não modifica o precondicionador. Os dados da Figura 7.33 mostram que houve redução do número de iterações quando utilizamos a melhor escolha de estratégias combinatórias, com destaque para reduções em 8 e 16 nós MPI.

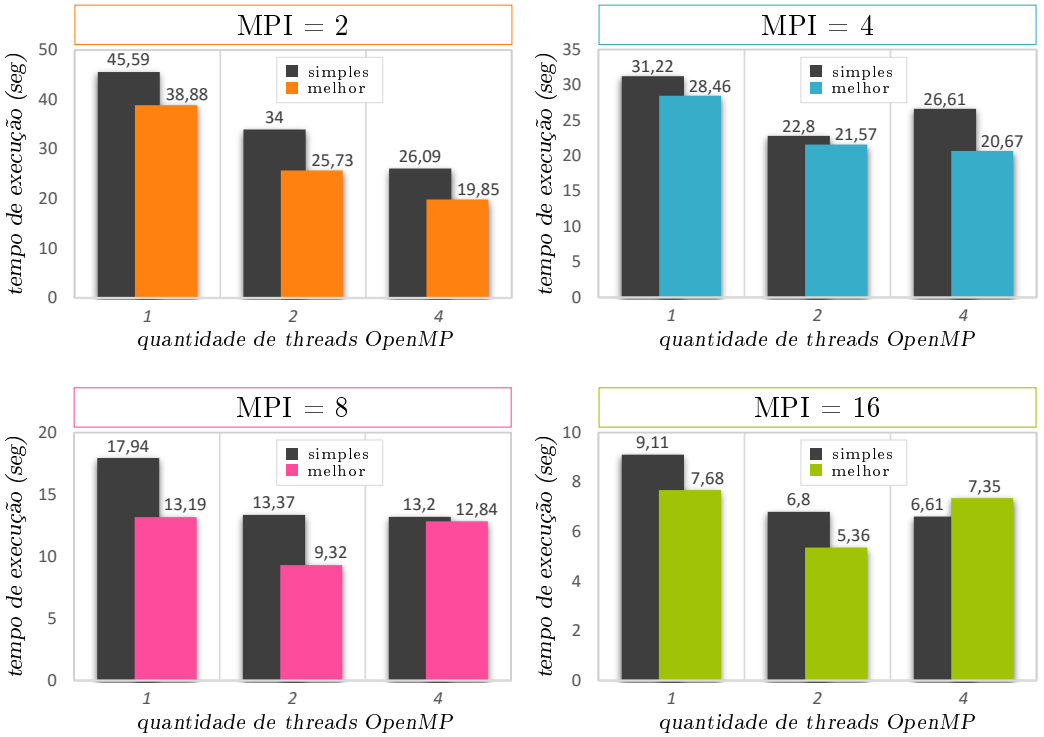


Figura 7.34: Escolha mais simples × melhor escolha de estratégias: Tempo (seg)

Os gráficos da Figura 7.34 comparam o tempo de execução da escolha mais simples com a melhor escolha de estratégias combinatórias, para todas as configurações de processadores apresentadas nas Tabelas 7.29 e 7.30.

Analisando esses gráficos podemos observar que, na maior parte dos casos, a melhor escolha de estratégias conseguiu reduzir o tempo computacional. A única exceção ocorreu em 64 (16×4) processadores onde o tempo da escolha mais simples foi melhor. Este fato pode estar relacionado a dimensão do problema em relação ao poder de processamento do *cluster*. Como os tempos em MPI=16 são pequenos, qualquer variação durante a execução do precondicionador pode influenciar nas medições desses tempos.

Concluímos que a escolha adequada de estratégias combinatórias possibilita uma boa redução do tempo de processamento. Porém, na prática, nem sempre podemos testar todas as combinações possíveis dessas estratégias. Nas matrizes de grande porte deste trabalho, onde o tempo de solução é alto, as escolhas de estratégias combinatórias foram feitas baseadas nos resultados observados nos testes com matrizes de menor porte.

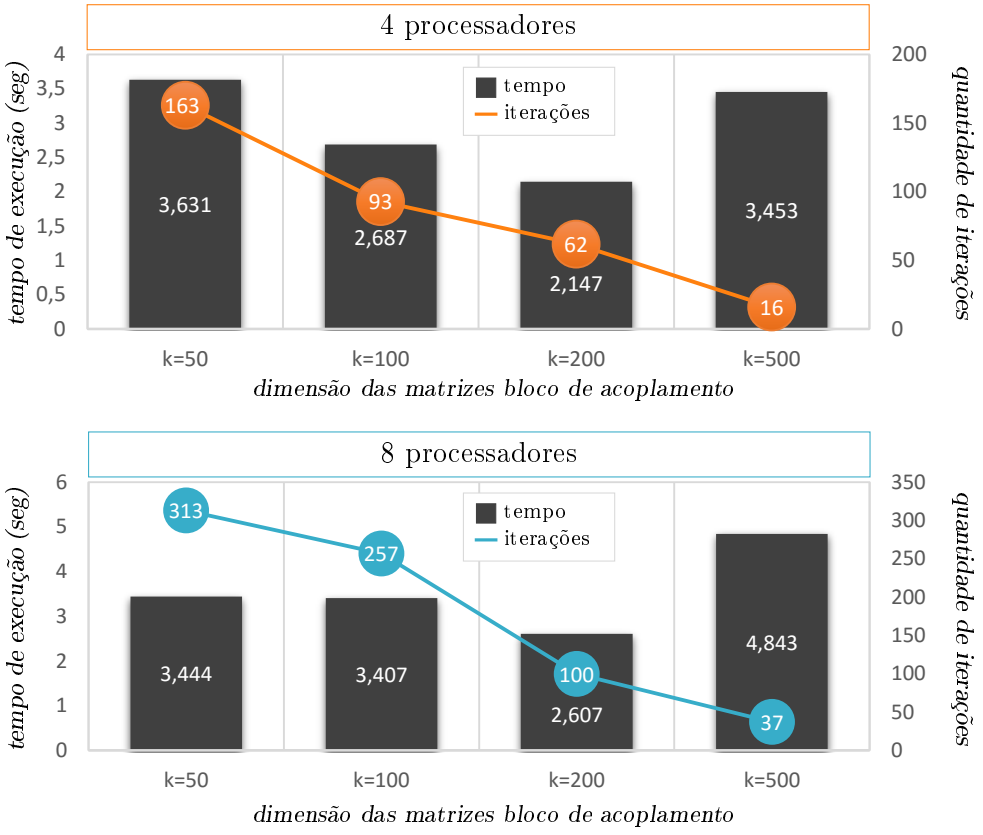
7.3 Influência do tamanho $\tilde{\mathbf{k}}$ das matrizes B_i e C_i

A dimensão $\tilde{\mathbf{k}}$ das matrizes bloco de acoplamento tem grande importância no tempo de solução e convergência do GMRES com precondicionador SPIKE. Quanto maior o valor de $\tilde{\mathbf{k}}$, mais elementos serão agrupados pelas matrizes B_i e C_i e, portanto, mais efetivo será o precondicionador. Em contrapartida, aumenta-se a complexidade dos sistemas da Equação 4.2 incluindo todas as operações que envolvem as matrizes *spikes* V_i e W_i, elevando o tempo de processamento. Logo, é necessário avaliar se aumentando o valor de $\tilde{\mathbf{k}}$, a redução do número de iterações do método iterativo irá compensar o tempo gasto pelo aumento da complexidade das operações do precondicionador.

A seguir, foram escolhidas duas matrizes com as mesmas configurações e parâmetros apresentadas na seção 7.1. Neste experimento, variamos o valor de $\tilde{\mathbf{k}}$ com 50, 100, 200 e 500 e a quantidade de processadores, entre 4, 8 e 16.

7.3.1 Matriz `rail_79841`

Os gráficos da Figura 7.35 mostram como a variação do $\tilde{\mathbf{k}}$ das matrizes bloco de acoplamento impacta no tempo de processamento e no número de iterações do GMRES, quando utilizamos 4, 8 e 16 processadores.



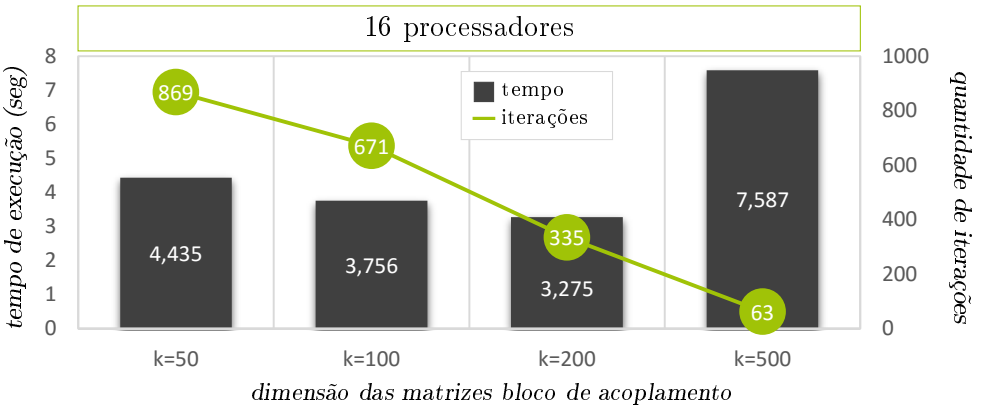


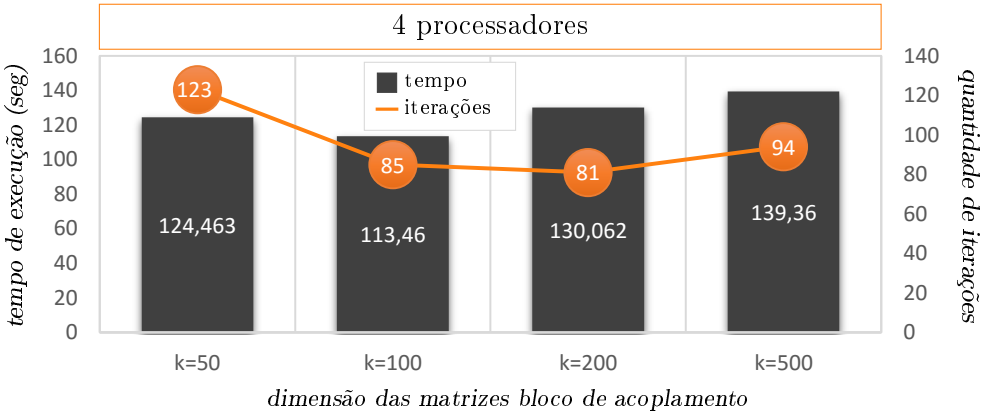
Figura 7.35: matriz `rail_79841`: Influência do tamanho de $\tilde{\mathbf{k}}$

Analisando os gráficos, podemos perceber um comportamento padrão para todas as configurações de processadores. Para $\tilde{\mathbf{k}} = 50$, $\tilde{\mathbf{k}} = 100$ e $\tilde{\mathbf{k}} = 200$, o número de iterações reduziu assim como o tempo de processamento do GMRES. Entretanto, para $\tilde{\mathbf{k}} = 500$, apesar da quantidade de iterações ter diminuído, o tempo de processamento aumentou consideravelmente. Isto que dizer que cada iteração do GMRES se tornou muito custosa computacionalmente que, mesmo com a redução acentuada no número de iterações, o tempo final da solução do sistema foi muito alto.

O melhor valor de $\tilde{\mathbf{k}}$ para essa matriz foi 200 para todas as configurações de processadores. Entretanto, como existe uma grande distância entre 200 e 500, é possível que se alcance melhores resultados variando o $\tilde{\mathbf{k}}$ entre esses dois valores.

7.3.2 Matriz `G3_circuit`

Da mesma maneira, os gráficos da Figura 7.36 mostram o impacto no tempo de processamento e no número de iterações do GMRES quando variamos o tamanho de $\tilde{\mathbf{k}}$ das matrizes bloco de acoplamento para 4, 8 e 16 processadores.



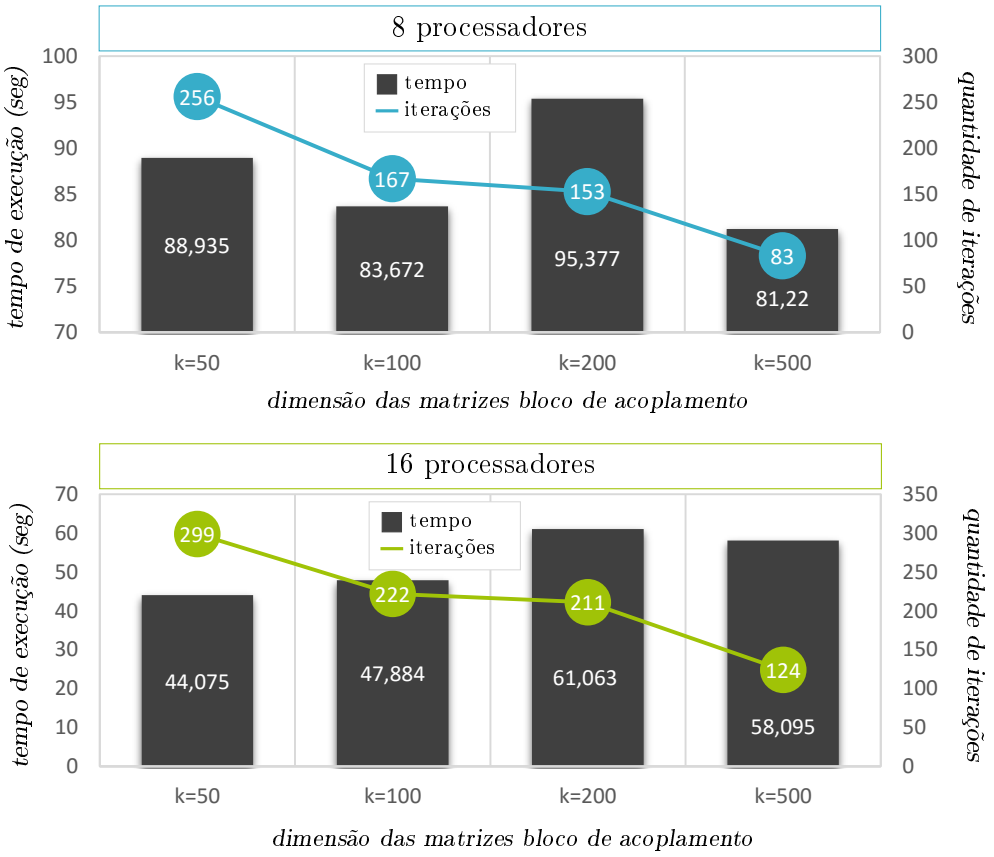


Figura 7.36: matriz `G3_circuit`: Influência do tamanho de $\tilde{\mathbf{k}}$

Analisando a Figura 7.36, o comportamento dos testes para a matriz `G3_circuit` foram bem distintos com 4, 8 e 16 processadores, diferente da matriz `rail_79841` cujo comportamento foi similar quando variamos o número de processadores (Figura 7.35).

Em 4 processadores, nem sempre houve queda na quantidade de iterações com o aumento dos valores de $\tilde{\mathbf{k}}$, como podemos ver em $\tilde{\mathbf{k}} = 500$ onde as iterações aumentam de 81 para 94. Em relação ao tempo de processamento, o melhor resultado obtido foi em $\tilde{\mathbf{k}} = 100$ com o menor tempo, passando a crescer com $\tilde{\mathbf{k}} = 200$ e $\tilde{\mathbf{k}} = 500$.

Analisando o gráfico com 8 processadores, observamos que o número de iterações reduziu para todo $\tilde{\mathbf{k}}$ e, ainda que essa redução não garanta a diminuição do tempo de processamento, o menor tempo ocorre quando $\tilde{\mathbf{k}} = 500$.

O comportamento apresentado no gráfico com 16 processadores mostra que o menor tempo de processamento foi em $\tilde{\mathbf{k}} = 50$, justamente onde o GMRES realizou o maior número de iterações. Esse fato mostra que as iterações com o $\tilde{\mathbf{k}} = 50$ foram muito mais econômicas e, adicionalmente, a redução do número de iterações alcançada com $\tilde{\mathbf{k}} = 100$, $\tilde{\mathbf{k}} = 200$ e $\tilde{\mathbf{k}} = 500$ não foi suficiente para compensar o custo de iterações mais custosas.

7.4 Influência do *cluster*

O experimento realizado nesta seção pretende mostrar o comportamento de execuções do precondicionador SPIKE em diferentes *clusters*. A matriz utilizada para os testes foi a `FEM_2D_1043474`, oriunda da aplicação descrita na subseção 7.5.1, cujas propriedades e parâmetros estão descritos abaixo.

Propriedades e Parâmetros	
Dimensão	1.043.474
Não-nulos	7.294.192
Estrutura	Não simétrica
Área de aplicação	Elementos Finitos
GMRES: Tolerância	10^{-8}
GMRES: *Restart* | Ciclos	50 | 1000
$\tilde{\mathbf{k}}$ (blocos de acoplamento)	50

Além do *cluster* Altix-xe, cujas configurações já foram apresentadas, os testes computacionais foram realizados em mais dois outros: o *cluster* Enterprise 3[1] da Universidade Federal do Espírito Santo e o *cluster* Gauss[2] da Universidade Federal do Rio Grande do Sul. O primeiro é um *cluster* mais antigo e, portanto, possui configurações modestas comparadas com *hardwares* atuais. Entretanto, optamos pela utilização deste *cluster* para alguns testes, pois tivemos fácil controle e suporte, além de dedicação exclusiva de seus recursos. O segundo é um *cluster* com configurações mais atuais, porém não foi dedicado aos nossos experimentos. Desta forma, os testes neste *cluster* sofreram grande oscilação na medição do tempo de execução.

O *cluster* Enterprise 3 conta com 24 nós Quad Core Intel 2 Q6600 (96 *cores*), com frequência de *clock* de 2.4GHz, 4MB de memória *cache* L2 e 4GB de RAM, interconectados com um *switch* Gigabit Ethernet 48-Port 4200G 3COM. O *cluster* Gauss opera com o Novell SUSE Linux Enterprise Server 11-SP1, e possui 64 unidades de processamento. Cada qual com 64 GB de RAM e dois processadores dodecacore AMD Opteron, totalizando 1.536 núcleos de processamento e um desempenho teórico de 15.97 TFlops. O padrão de conexão da rede é InfiniBand.

Quanto as especificações de *software*, os códigos nos *clusters* Enterprise 3 e Gauss foram compilados com GCC 4.8.4 e, no Altix-xe, com compiladores Intel. É importante dizer que não desejamos realizar comparações entre *clusters* pois suas especificações de *hardware* e *software* são completamente diferentes. Entretanto, desejamos estudar o comportamento do nosso algoritmo com essas diferentes plataformas e configurações.

Todas as tabelas a seguir mostram o tempo de execução em segundos e quantidade de iterações de execuções com até 64 processadores, variando a quantidade de nós MPI e *threads* OpenMP. As figuras, após cada tabela, mostram o gráfico de *speedup*.

[1] enterprise3.lcad.inf.ufes.br/ganglia
[2] http://www.cesup.ufrgs.br/recursos-e-servicos/hardware-1/sgi-altix

7.4.1 Altix-xe (LNCC)

Os testes a seguir foram realizados no *cluster* Altix-xe do Laboratório Nacional de Computação Científica (LNCC) localizado em Petrópolis, no Rio de Janeiro. Este *cluster* foi utilizado na fase final deste trabalho, a partir da necessidade de executar nossos experimentos em uma máquina com configurações mais modernas.

		OpenMP		
MPI	Iterações	1	2	4
2	20	38,88	25,73	19,85
4	33	28,46	21,57	20,67
8	39	13,19	9,32	12,84
16	56	7,68	5,36	7,35

Tabela 7.31: Tempos (seg) no *cluster* Altix-xe (LNCC)

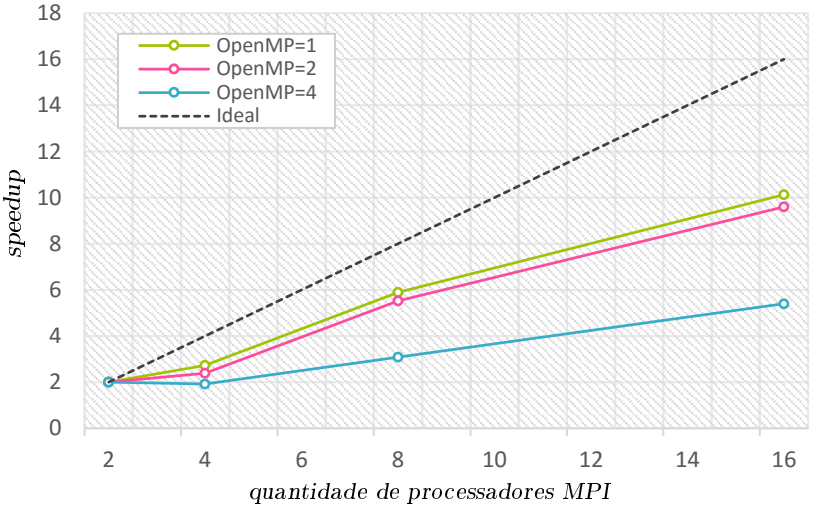


Figura 7.37: *Speedup* do *cluster* Altix-xe (LNCC)

A Tabela 7.31 mostra um boa redução no tempo de processamento quando aumentamos o número de nós MPI e *threads* OpenMP. Neste caso, o melhor tempo foi obtido em 32 processadores na configuração 16×2. Como podemos observar, nas configurações com 8 e 16 nós MPI, o tempo com 2 *threads* foi melhor do que com 4. Isso porque, o alto poder de processamento desse *cluster* para essa dimensão de problema faz com que, em cada processador, o tempo para resolver sua respectiva tarefa fosse muito pequeno, evidenciando o tempo gasto com comunicação.

O *speedup* da figura 7.37 mostra um comportamento muito similar quando utilizamos 1 ou 2 *threads* OpenMP, sendo sutilmente melhor com 1 *thread*. Observando 4 *threads*, o *speedup* se mostrou muito inferior as demais configurações.

7.4.2 Enterprise 3 (UFES)

O Enterprise 3, da Universidade Federal do Espírito Santo, é um *cluster* mais antigo e com um hardware limitado. Entretanto, foi bastante utilizado na fase inicial dos testes pelo fácil acesso e utilização dedicada à aplicação deste trabalho.

		OpenMP		
MPI	Iterações	1	2	4
2	20	71,90	52,32	44,24
4	33	37,50	29,44	26,55
8	39	18,37	14,76	13,60
16	55	11,30	9,46	8,77

Tabela 7.32: Tempos (seg) no *cluster* Enterprise 3 (UFES)

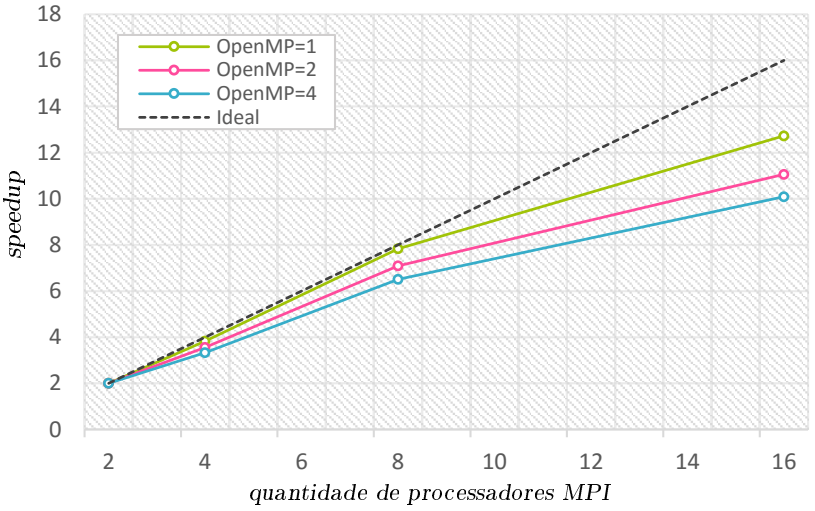


Figura 7.38: *Speedup* do *cluster* Enterprise 3 (UFES)

Neste *cluster*, os tempos de execução decaem com o aumento dos nós MPI e de *threads* OpenMP, como podemos observar na Tabela 7.32. Diferente do *cluster* da seção anterior, este possui um *hardware* modesto fazendo com que as execuções demorem mais tempo, evidenciando o ganho com a divisão de tarefas. O melhor tempo ocorreu com 64 processadores, na configuração 16×4.

Na Figura 7.38 observamos que o *speedup* alcançado no Enterprise 3 é bem parecido ao serem consideradas 1, 2 ou 4 *threads* OpenMP, apresentando uma boa escalabilidade. A melhor configuração de escalabilidade foi, mais uma vez, quando utilizamos apenas 1 *thread* OpenMP.

7.4.3 Gauss (UFRGS)

Devido a necessidade de realizar os testes em uma máquina mais potente, o *cluster* Gauss, localizado na Universidade Federal do Rio Grande do Sul, foi utilizado. Entretanto, esse *cluster* é intensamente procurado por outros usuários, fazendo com que nossos experimentos ficassem muito tempo na fila de espera, aguardando recursos disponíveis para que pudessem ser executados.

		OpenMP		
MPI	Iterações	1	2	4
2	20	90,70	121,44	114,40
4	33	324,51	389,25	288,39
8	39	261,68	341,36	150,76
16	53	183,35	333,34	333,04

Tabela 7.33: Tempos (seg) no *cluster* Gauss (UFRGS)

A Tabela 7.33 mostra que os tempos de execução no *cluster* Gauss não apresentaram um comportamento esperado isto é, de redução do tempo de processamento com o aumento do número de processadores. Além disso, aferidos tempos não seguiram nenhum padrão de redução ou aumento. Conjecturamos que este comportamento se deve ao fato do *cluster* Gauss não ter sido dedicado aos nossos experimentos, ao contrário dos outros dois primeiros *cluster*. A grande demanda de recursos do *cluster* Gauss — como uso de CPU e memória — exigida por processos de outros usuários pode ter impactado fortemente nos tempos de execução dos nossos testes. Como não houve *speedup*, o respectivo gráfico não é apresentado neste teste. Como podemos perceber, o uso deste *cluster* apresentou um resultado extremamente desfavorável à essa aplicação.

7.5 *Speedup*

Nesta seção, mostramos duas aplicações utilizando o método de elementos finitos cujos domínios foram discretizados por elementos triangulares — para o caso 2D — e elementos tetraédricos — para o caso 3D.

O objetivo é estudar detalhadamente o comportamento do precondicionador SPIKE, em um conjunto de testes realizados no *cluster* Altix-xe, à medida que variamos as configurações de processadores distribuídos e compartilhados. Para isso, avaliamos o *speedup* e a escalabilidade desse precondicionador em cada uma das aplicações a seguir.

7.5.1 Aplicação de Elementos Finitos 2D

Para este experimento, consideramos um problema bidimensional de referência descrito pela equação difusiva-advectiva

$$\boldsymbol{\beta}.\nabla u - \nabla.(\boldsymbol{\kappa}\nabla u) = f, \quad \text{em} \quad \Omega = [-1,1] \times [-1,1] \ , \tag{7.1}$$

onde u representa a quantidade sendo transportada (ex: temperatura, concentração). Para este problema, considere $\boldsymbol{\kappa} = 10^{-7}I$, $\boldsymbol{\beta} = (-y, x)^T$ e $f = 0$ satisfazendo as condições de contorno homogêneas, ou seja, $u(x, y) = 0$ para $(x, y) \in \Gamma$. Supondo o ponto $O = (0, 0)$ e o ponto $A = (0, -1)$, o segmento OA é uma fronteira interna com valores prescritos, $u(x, y) = u_0(0, y) = sen(\pi y)$, como mostra a Figura 7.39.

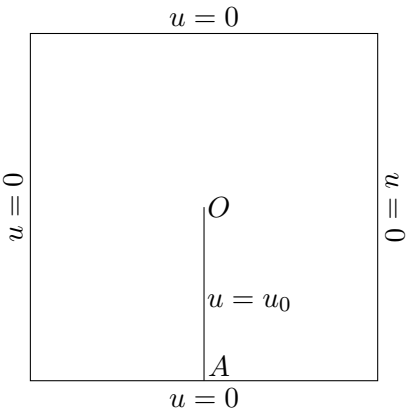


Figura 7.39: Condições de contorno 2D

Foi gerada uma malha pelo *software* GMSH [Geuzaine and Remacle, 2009] de forma a se obter uma matriz de grande porte. A matriz deste teste é a `FEM_2D_3381977` apresentada na subseção 7.1.11 com as mesmas estratégias combinatórias.

A Tabela 7.34 mostra os tempos para algumas configurações de nós MPI e *threads* OpenMP. Infelizmente, não foi possível testar algumas configurações de processadores

distribuídos e *threads* devido a limitações de 96 processadores por usuário ou de quantidade de nós distribuídos maior que 16, destacados na tabela por >96 e >16, respectivamente, segundo regras estabelecidas pelo *cluster* Altix-xe do LNCC.

		OpenMP			
MPI	Iterações	1	2	4	8
4	44	133,560	107,888	90,295	104,376
8	50	67,554	41,063	58,497	102,836
12	69	59,698	33,359	46,101	59,756
16	74	38,665	27,112	39,691	>96
24	80	26,137	>16	>16	>96
48	111	16,894	>16	>96	>96
64	132	11,738	>96	>96	>96

Tabela 7.34: Tempos (seg) para diferentes configurações de processadores

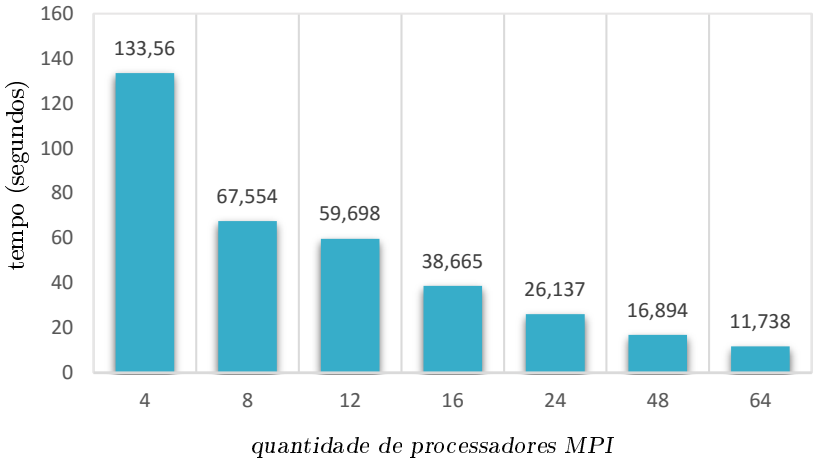


Figura 7.40: Tempo (seg) para diferentes nós MPI e 1 *thread*

Observando a Tabela 7.34, vemos que nem sempre o aumento do número de *threads* causou a redução do tempo de processamento, como podemos ver em 4 e 8 *threads*. Apesar de não ter sido possível executar todos os testes com 2 *threads*, os melhores tempos ocorreram nessa configuração. Já o aumento da quantidade de nós MPI sempre teve impacto positivo no tempo final e, mesmo aumentando o número de iterações, foi capaz de alcançar apenas 16,894 segundos quando utilizamos 48 processadores e 1 *thread*. Uma análise interessante é que, para mesma quantidade de processadores, o aumento dos nós MPI tem impacto mais significativo do que o aumento de *threads*, por exemplo, o tempo com 64 processadores na configuração 64×1 é melhor do que o tempo com a mesma quantidade de processadores, porém na configuração 16×4.

A Figura 7.40 mostra uma boa redução do tempo de execução do GMRES com precondicionador SPIKE à medida que aumentamos a quantidade de processadores MPI, considerando os testes com 1 *thread*.

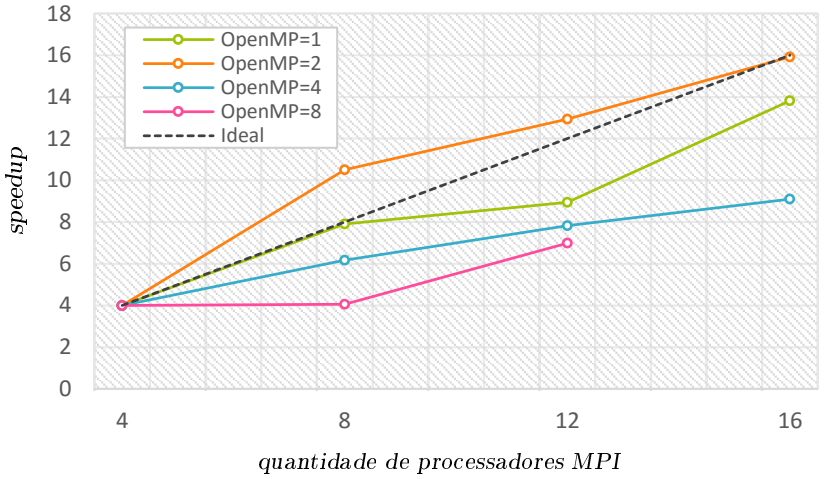


Figura 7.41: *Speedup* de 1, 2, 4 e 8 *threads* OpenMP

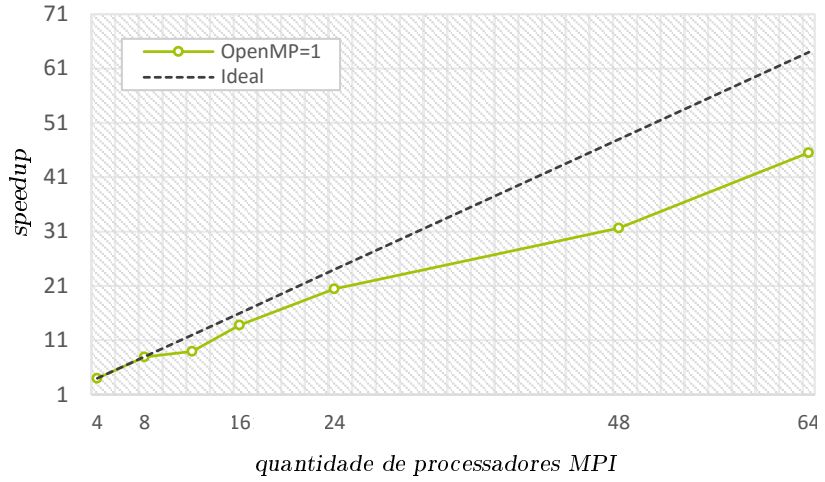


Figura 7.42: *Speedup* para diferentes nós MPI e 1 *thread*

O gráfico da Figura 7.41 exibe o *speedup* das *threads* para 4, 8, 12 e 16 nós MPI. Nele, observamos um *speedup* bom para 1 e 2 *threads*, e razoável para 4 e 8. Vale destacar que o melhor resultado ocorreu com 2 *threads*, com um *speedup* superlinear inicialmente. Umas das possíveis razões deste *speedup* superlinear está no uso otimizado da memória *cache* dado a dimensão do problema naquela específica configuração, isto é, grande parte da carga de trabalho pode ter sido carregada na *cache*, reduzindo drasticamente o tempo de acessos à memória RAM. No geral, o gráfico sugere uma boa escalabilidade.

A Figura 7.42 apresenta o *speedup* de 1 *threads* para todos os processadores MPI testados. Novamente, podemos observar uma boa escalabilidade com *speedup* linear para 4 e 8 nós MPI, decaindo ao longo dos demais processadores.

7.5.2 Aplicação de Elementos Finitos 3D

Este é um problema tridimensional de transferência de calor definido pela equação

$$-\nabla.(\boldsymbol{\kappa}\nabla u) = 0, \quad \text{em} \quad \Omega = [0,1] \times [0,1] \times [0,1] \tag{7.2}$$

onde u representa a temperatura e $\boldsymbol{\kappa} = I$, a condutividade térmica. As condições de contorno deste problema são definidas como $u(x, y, 0) = u(x, 0, z) = u(1, y, z) = 0$ e $u(x, y, 1) = u(x, 1, z) = u(0, y, z) = 100$, como mostra a Figura 7.43.

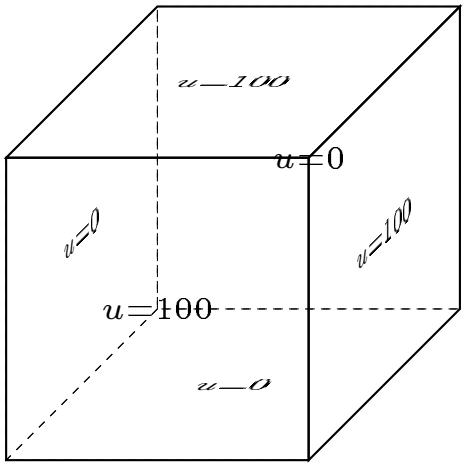


Figura 7.43: Condições de contorno 3D

Para este problema geramos uma malha pelo *software* GMSH, obtendo a matriz `FEM_3D_938586` da subseção 7.1.12 com as mesmas estratégias combinatórias.

A Tabela 7.35 mostra os tempos para algumas configurações de nós MPI e *threads* OpenMP. É importante relembrar que não foi possível testar algumas configurações de processadores distribuídos e *threads* devido a limitações de 96 processadores por usuário, quantidade de nós distribuídos maior do que 16 e insuficiência quanto a quantidade de memória RAM, destacados na tabela por >96, >16 e RAM, respectivamente.

		OpenMP			
MPI	Iterações	1	2	4	8
4	71	833,668	426,167	524,099	RAM
8	94	237,396	131,668	153,740	239,637
12	146	150,090	82,978	90,217	159,644
16	274	124,466	79,269	88,738	>96
24	256	73,582	>16	>16	>96
30	250	45,679	>16	>96	>96

Tabela 7.35: Tempos (seg) para diferentes configurações de processadores

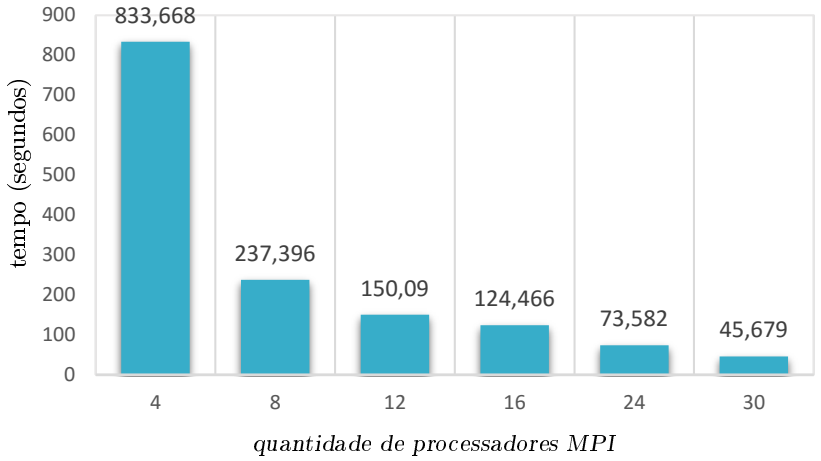


Figura 7.44: Tempo (seg) para diferentes nós MPI e 1 *thread*

A Tabela 7.35 apresenta um comportamento parecido ao do teste anterior onde obtemos uma boa redução do tempo de processamento com o aumento de processadores distribuídos. O melhor tempo acontece em 30 processadores e 1 *thread*, com tempo de execução igual a 45,679 segundos em 250 iterações do GMRES. A configuração 4×8 não foi executada devido a falta de memória RAM para a etapa de fatoração numérica do PARDISO. A divisão dos 24GB de RAM em 8 *threads*, em cada processador, determina que cada *thread* tenha 3G à disposição, o que não foi suficiente considerando uma partição dos dados em apenas 4 processadores distribuídos. Testes com mais de 30 processadores MPI também não foram possíveis devido a restrições no produto matriz-vetor, i.e., nem todos elementos não nulos foram cobertos pelos blocos diagonais e de acoplamento, após o particionamento. Um fato incomum é que a quantidade de iterações diminuiu em 24 e 30 processadores.

Na Figura 7.44 observamos uma grande redução do tempo de processamento ao longo dos processadores MPI, com uma queda expressiva de 4 para 8 processadores.

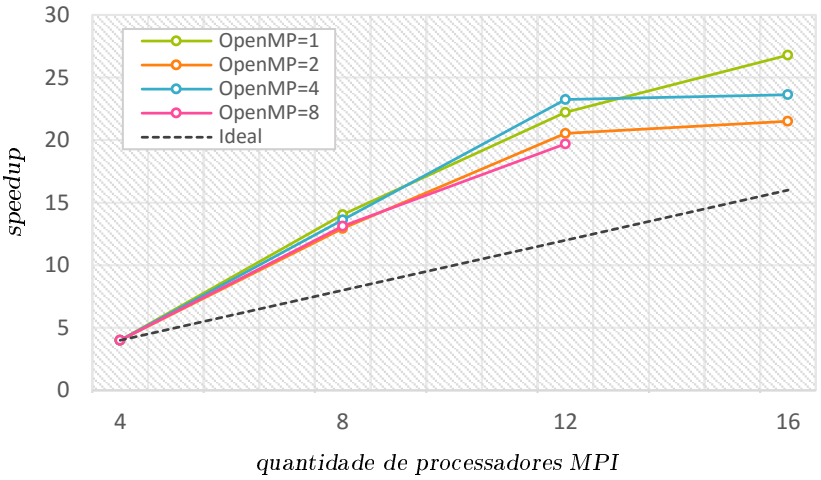


Figura 7.45: *Speedup* de 1, 2, 4 e 8 *threads* OpenMP

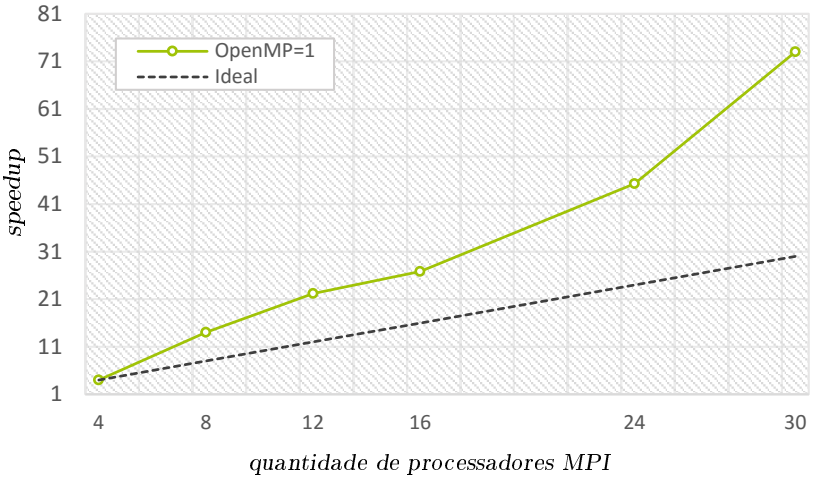


Figura 7.46: *Speedup* para diferentes nós MPI e 1 *thread*

Para plotar a curva de *speedup* de 8 *threads*, na Figura 7.45, o tempo na configuração 4×8 foi estimado proporcionalmente em relação aos demais valores da tabela. Analisando esta tabela, podemos observar um *speedup* superlinear para todas as configurações de *threads*. Além do uso otimizado da memória *cache*, uma outra possível explicação é o fato de que o algoritmo SPIKE se mostrou escalável para 3.072 *cores* em Naumann and Schenk [2012], um número de processadores muito superior aos apresentados neste teste. Os melhores *speedups* foram para OpenMP=1 e OpenMP=4.

A Figura 7.46 também mostra um *speedup* superlinear para 1 *thread* e até 30 nós MPI, sugerindo uma excelente escalabilidade.

7.6 Análise de Desempenho

O comportamento dos nossos algoritmos foi estudado utilizando o *software* TAU (*Tuning and Analysis Utilities*) [Shende and Malony, 2006], que contém um conjunto de ferramentas para análise de desempenho de programas paralelos de diversas linguagens. A Figura 7.47 exibe o tempo gasto — em ordem decrescente — das principais funções em uma execução do GMRES com precondicionador SPIKE utilizando a matriz `FEM_2D_1043474`, para 8 processadores MPI e apenas 1 *thread*.

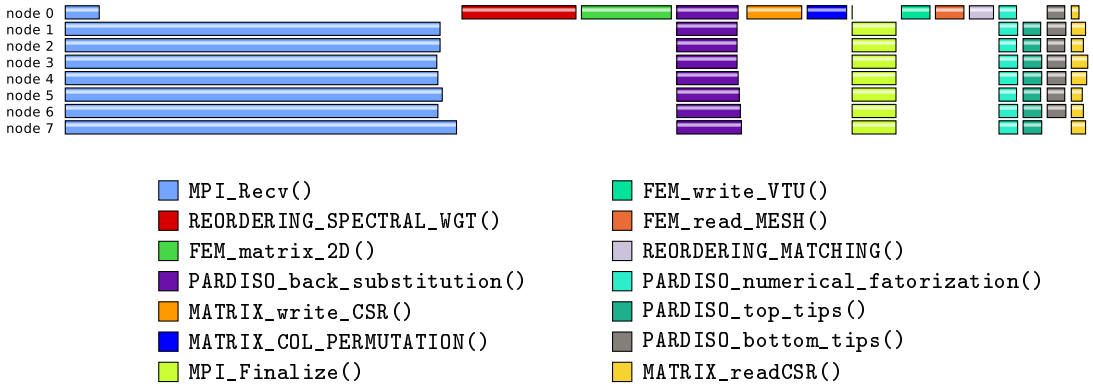


Figura 7.47: TAU: Funções com maior tempo de processamento (barras separadas)

Analisando essa figura podemos perceber que a operação mais custosa é, de fato, a função `MPI_Recv()` que realiza a comunicação entre processadores. Muitas funções seriais aparecem sendo feitas no *rank* 0, como o reordenamento Espectral Valorado (`REORDERING_SPECTRAL_WGT()`), o *matching* (`REORDERING_MATCHING()`) e algumas funções de leitura, escrita e montagem da matriz de elementos finitos. A segunda operação paralela responsável pelo maior consumo de tempo de processamento é a `PARDISO_back_substitution()` que é executada duas vezes a cada iteração pelo Pardiso para encontrar a solução de um sistema através de substituições sucessivas. Outras operações paralelas do *software* Pardiso tais como `PARDISO_top_tips()`, `PARDISO_top_tips()` e `PARDISO_numerical_fatorization()` consomem uma pequena parte do tempo total de processamento.

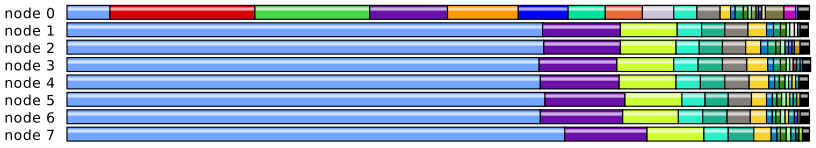


Figura 7.48: TAU: Funções com maior tempo de processamento (barras agrupadas)

A Figura 7.48 apresenta os mesmos dados da Figura 7.47, porém com as barras agrupadas. Nela podemos observar que, ainda que o *rank* 0 tenha feito mais operações seriais, isto não comprometeu o tempo total, uma vez que esse *rank* não realiza muitas operações de troca de mensagens. O tempo gasto por cada *rank* é muito similar, sugerindo que a carga de trabalho paralela está bem distribuída. Vale ressaltar que a figura em questão possui algumas funções cujos tempos de execução foram pouco representativos no tempo de solução e, por isso, não foram mostradas na Figura 7.47.

Capítulo 8

Conclusão e Trabalhos Futuros

Neste trabalho, apresentamos um algoritmo paralelo híbrido, SPIKE, eficiente quando utilizado como um precondicionador para o método iterativo GMRES na resolução de sistemas lineares. Testes computacionais mostraram que o precondicionador SPIKE foi capaz de reduzir consideravelmente a quantidade de iterações e o tempo de processamento dentro do método iterativo. Além disso, matrizes mal condicionadas passaram a convergir após o uso desse precondicionador.

Significativo desempenho e escalabilidade foram obtidos, de modo geral, nos testes com múltiplos processadores distribuídos e compartilhados, apesar da limitação referente a quantidade de processadores. O uso das bibliotecas MPI e OpenMP para o esquema híbrido de arquitetura paralela se mostrou vantajoso na redução do tempo de CPU, como esperado. Nas matrizes de grande porte obtidas das aplicações de elementos finitos, o uso de múltiplos processadores foi capaz de reduzir — no melhor caso — cerca de 91% do tempo de processamento do problema bidimensional e, aproximadamente, 96% do tempo de processamento do problema tridimensional.

O uso de estratégias combinatórias também se mostrou bastante eficaz em melhorar o desempenho do algoritmo SPIKE. As técnicas baseadas em algoritmos em grafos realizam transformações nas matrizes que levam-na a adquirir propriedades convenientes para a utilização do SPIKE, reduzindo os custos do método iterativo e o tempo de solução final. Conforme observamos nos experimentos, a utilização de todas as estratégias combinatórias, ao mesmo tempo, nem sempre garante um melhor precondicionador. Por este motivo, no geral, é preciso conhecer as características da matriz e, se possível, realizar um conjunto de testes para escolher a melhor combinação dessas estratégias. Uma observação importante foi com relação ao alto tempo de execução de algumas estratégias combinatórias, com destaque para o reordenamento Espectral e Espectral Valorado, que muitas vezes representou grande parte do tempo total de solução. Como trabalhos futuros, uma boa direção é o estudo e implementação desses algoritmos usando programação paralela. O artigo de Manguoglu et al. [2011]

apresenta um algoritmo paralelo para cálculo do vetor de *Fiedler*, a etapa mais custosa dos algoritmos de Espectral e Espectral Valorado. O estudo de Karantasis et al. [2014] desenvolve uma implementação paralela dos algoritmos RCM e Sloan, apresentando bons resultados em testes com até 16 processadores.

Os experimentos computacionais nos fizeram perceber uma limitação do nosso código, com relação a implementação do produto matriz-vetor. Apesar das otimizações aplicadas ao SpMV, o esquema escolhido neste trabalho apresentou algumas desvantagens.

A primeira é referente a utilização da memória *cache*. Em cada passo do SpMV, as entradas da matriz são utilizadas apenas uma vez e, por outro lado, os vetores densos x e y são usados múltiplas vezes. Ao lidarmos com problemas de grande porte, em especial quando a ordem da matriz e dos vetores é muito maior do que o tamanho da memória *cache*, os mesmos dados serão, eventualmente, excluídos e recuperados de volta para *cache* causando perda de desempenho. O artigo escrito por Yzelman et al. [2009] expõe detalhadamente o comportamento da memória *cache* no SpMV paralelo, considerando estruturas de armazenamento otimizadas, e propõe um particionamento baseado em hipergrafos para reduzir a quantidade de *cache misses*. Sosonkina et al. [2007] também apresenta um conjunto de técnicas de particionamentos baseados em hipergrafos, em especial o particionamento Mondriaan, analisando suas diversas vantagens, tais como, a redução de comunicação alcançada e a habilidade de lidarem com matrizes não simétricas ou que apresentam estruturas mais complexas.

A segunda desvantagem do esquema de particionamento escolhido para o SpMV é a perda da generalidade, i.e., dependendo da matriz utilizada e da quantidade de divisões em processadores distribuídos, não é possível realizar o particionamento. Este fato acontece porque, para algumas matrizes, o reordenamento não é capaz de reduzir sua largura de banda a fim de manter todos os elementos não nulos agrupados dentro dos blocos A_i, $\hat{C}_i$ e $\hat{B}_i$. Se, após o reordenamento, os blocos não cobrirem todos os elementos, então não será possível realizar o SpMV para essas matrizes, uma vez que esses elementos não serão considerados. A solução é implementar essa operação de maneira mais geral, utilizando bibliotecas ou códigos já disponíveis na literatura como os da PETSc (*Portable, Extensible Toolkit for Scientific Computation*) disponível em [Balay et al., 2015].

Como trabalhos futuros, além da paralelização dos algoritmos seriais, sugerimos o estudo e inclusão de novas estratégias combinatórias que fortaleçam o efeito do precondicionador SPIKE. Além disso, é extremamente importante avaliar seu desempenho e escalabilidade em *clusters* com um número de processadores maior do que foi possível neste trabalho.

Referências Bibliográficas

Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, and Hong Zhang. PETSc Web page. http://www.mcs.anl.gov/petsc, 2015. URL http://www.mcs.anl.gov/petsc.

S. T. Barnard, A. Pothen, and H. D. Simon. A spectral algorithm for envelope reduction of sparse matrices. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Supercomputing '93, pages 493–502, New York, NY, USA, 1993. ACM. ISBN 0-8186-4340-4. doi: 10.1145/169627.169790. URL http://doi.acm.org/10.1145/169627.169790.

Blaise Barney. *Introduction to Parallel Computing*. Lawrence Livermore National Laboratory, 09 2007. URL https://computing.llnl.gov/tutorials/parallel_comp/.

Michele Benzi, Daniel B. Szyld, and Arno Van Duin. Orderings for incomplete factorization preconditioning of nonsymmetric problems. *SIAM J. SCI. COMPUT*, 20(5):1652–1670, 1999.

Ming-yu Chen, Lily Mummert, Padmanabhan Pillai, Alexander Hauptmann, and Rahul Sukthankar. Exploiting multi-level parallelism for low-latency activity recognition in streaming video. In *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems*, MMSys '10, pages 1–12, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-914-5. doi: 10.1145/1730836.1730838. URL http://doi.acm.org/10.1145/1730836.1730838.

S. C. Chen, D. J. Kuck, and A. H. Sameh. Practical parallel band triangular system solvers. *ACM Trans. Math. Softw.*, 4(3):270–277, September 1978. ISSN 0098-3500. doi: 10.1145/355791.355797. URL http://doi.acm.org/10.1145/355791.355797.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848. Section 24.3: Dijkstra's algorithm.

E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, ACM '69, pages 157–172, New York, NY, USA, 1969. ACM. doi: 10.1145/800195.805928. URL http://doi.acm.org/10.1145/800195.805928.

T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, (1):1:1 – 1:25, 2011. URL http://www.cise.ufl.edu/research/sparse/matrices.

T. A. Davis, P. Amestoy, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17:886–905, 1994.

Peter J Denning and Walter F Tichy. Highly parallel computation. Technical report, Research Inst. for Advanced Computer Science; Moffett Field, CA, United States, 1990.

Jack J. Dongarra and Ahmed H. Sameh. On some parallel banded system solvers. *Parallel Comput.*, 1(3-4):223–235, December 1984. ISSN 0167-8191. doi: 10.1016/S0167-8191(84)90165-0. URL http://dx.doi.org/10.1016/S0167-8191(84)90165-0.

Iain S. Duff and Jacko Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Anal. Appl.*, 20(4):889–901, July 1999. ISSN 0895-4798. doi: 10.1137/S0895479897317661. URL http://dx.doi.org/10.1137/S0895479897317661.

Iain S. Duff and Jacko Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. Matrix Anal. Appl.*, 22(4):973–996, July 2000. ISSN 0895-4798. doi: 10.1137/S0895479899358443. URL http://dx.doi.org/10.1137/S0895479899358443.

C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, DAC '82, pages 175–181, Piscataway, NJ, USA, 1982. IEEE Press. ISBN 0-89791-020-6. URL http://dl.acm.org/citation.cfm?id=800263.809204.

Miroslav Fiedler. Algebraic connectivity of graphs. *Czechoslovak Mathematical Journal*, 23:298–305, 1973.

Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973. doi: 10.2307/2156361. URL http://dx.doi.org/10.2307/2156361.

Alan George and Esmond Ng. Symbolic factorization for sparse gaussian elimination with partial pivoting. *SIAM J. Sci. Stat. Comput.*, 8(6):877–898, November 1987. ISSN 0196-5204. doi: 10.1137/0908072. URL http://dx.doi.org/10.1137/0908072.

Christophe Geuzaine and Jean-François Remacle. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *Int. J. Numer. Meth. Engng.*, 79(11):1309–1331, September 2009. ISSN 00295981. doi: 10.1002/nme.2579. URL http://dx.doi.org/10.1002/nme.2579.

Kamila Ribeiro Ghidetti. O impacto do reordenamento de matrizes esparsas nos métodos iterativos não estacionários precondicionados. Master's thesis, Universidade Federal do Espírito Santo, 2011.

N. E. Gibbs, W. G Poole, and P. K. Stockmeyer. An algorithm for reducing the profile and bandwidth of a sparse matrix. *SIAM J. Numer. Anal.*, 13:236–250, 1976.

Tífani Teixeira Gonçalez. Algoritmos adaptativos para o método GMRES(m). Master's thesis, Universidade Federal do Rio Grande do Sul, 2005.

Dana A. Jacobsen and Inanc Senocak. Multi-level parallelism for incompressible flow computations on gpu clusters. *Parallel Comput.*, 39(1):1–20, January 2013. ISSN 0167-8191. doi: 10.1016/j.parco.2012.10.002. URL http://dx.doi.org/10.1016/j.parco.2012.10.002.

Konstantinos I. Karantasis, Andrew Lenharth, Donald Nguyen, María J. Garzarán, and Keshav Pingali. Parallelization of reordering algorithms for bandwidth and wavefront reduction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 921–932, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-5500-8. doi: 10.1109/SC.2014.80. URL http://dx.doi.org/10.1109/SC.2014.80.

Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Commun. ACM*, 33(5):539–543, May 1990. ISSN 0001-0782. doi: 10.1145/78607.78614. URL http://doi.acm.org/10.1145/78607.78614. Tradução minha.

B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970. ISSN 1538-7305. doi: 10.1002/j.1538-7305.1970.tb01770.x. URL http://dx.doi.org/10.1002/j.1538-7305.1970.tb01770.x.

A. Kuzmin, M. Luisier, and O. Schenk. Fast methods for computing selected elements of the greens function in massively parallel nanoelectronic device simulations. In F. Wolf, B. Mohr, and D. Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *LNCS*, pages 533–544. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-40046-9. doi: 10.1007/978-3-642-40047-6_54. URL http://dx.doi.org/10.1007/978-3-642-40047-6_54.

Josep-Lluis Larriba-Pey, Angel Jorba, and Juan J. Navarro. Spike algorithm with savings for strictly diagonal dominant tridiagonal systems. *Microprocess. Microprogram.*, 39

(2-5):125–128, December 1993. ISSN 0165-6074. doi: 10.1016/0165-6074(93)90071-R. URL http://dx.doi.org/10.1016/0165-6074(93)90071-R.

D H. Lawrie and A H. Sameh. The computation and communication complexity of a parallel banded system solver. *ACM Trans. Math. Softw.*, 10(2):185–195, May 1984. ISSN 0098-3500. doi: 10.1145/399.401. URL http://doi.acm.org/10.1145/399.401.

Brenno Lugon. Algoritmos de reordenamento de matrizes esparsas aplicados a precondicionadores ILU(p). *Simpósio Brasileiro de Pesquisa Operacional (SBPO)*, 45: 2343–2355, 2013.

Murat Manguoglu, Ahmed H. Sameh, and Olaf Schenk. PSPIKE: A parallel hybrid sparse linear system solver. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704 of *LNCS*, pages 797–808. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-03868-6. doi: 10.1007/978-3-642-03869-3_74. URL http://dx.doi.org/10.1007/978-3-642-03869-3_74.

Murat Manguoglu, Mehmet Koyutürk, Ahmed H. Sameh, and Ananth Grama. Weighted matrix ordering and parallel banded preconditioners for iterative linear system solvers. *SIAM J. Sci. Comput.*, 32(3):1201–1216, April 2010. ISSN 1064-8275. doi: 10.1137/080713409. URL http://dx.doi.org/10.1137/080713409.

Murat Manguoglu, Eric Cox, Faisal Saied, and Ahmed Sameh. Tracemin-fiedler: A parallel algorithm for computing the fiedler vector. In JoséM.LaginhaM. Palma, Michel Daydé, Osni Marques, and JoãoCorreia Lopes, editors, *High Performance Computing for Computational Science – VECPAR 2010*, volume 6449 of *Lecture Notes in Computer Science*, pages 449–455. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-19327-9. doi: 10.1007/978-3-642-19328-6_40. URL http://dx.doi.org/10.1007/978-3-642-19328-6_40.

Fredrik Manne and Tor Sørevik. Optimal partitioning of sequences. *J. Algorithms*, 19 (2):235–249, September 1995. ISSN 0196-6774. doi: 10.1006/jagm.1995.1035. URL http://dx.doi.org/10.1006/jagm.1995.1035.

MKL. *Intel Math Kernel Library Reference Manual*. Intel Corporation, 2013. ISBN 630813-054US. URL https://software.intel.com/sites/default/files/managed/c8/36/mklman_c_11.3.pdf.

Uwe Naumann and Olaf Schenk. *Combinatorial Scientific Computing*. Chapman & Hall/CRC, Chapter 4, 1st edition, 2012. ISBN 1439827354, 9781439827352.

Ch.H. Papadimitriou. The np-completeness of the bandwidth minimization problem. *Computing*, 16(3):263–270, 1976. ISSN 0010-485X. doi: 10.1007/BF02280884. URL http://dx.doi.org/10.1007/BF02280884.

Ali Pinar and Cevdet Aykanat. Fast optimal load balancing algorithms for 1d partitioning. *J. Parallel Distrib. Comput.*, 64(8):974–996, August 2004. ISSN 0743-7315. doi: 10.1016/j.jpdc.2004.05.003. URL http://dx.doi.org/10.1016/j.jpdc.2004.05.003.

Eric Polizzi and Ahmed H. Sameh. A parallel hybrid banded system solver: The SPIKE algorithm. *Parallel Comput.*, 32(2):177–194, February 2006. ISSN 0167-8191. doi: 10.1016/j.parco.2005.07.005. URL http://dx.doi.org/10.1016/j.parco.2005.07.005.

Eric Polizzi and Ahmed H. Sameh. SPIKE: A parallel environment for solving banded linear systems. *Computers & Fluids*, 36(1):113–120, January 2007. ISSN 00457930. doi: 10.1016/j.compfluid.2005.07.005. URL http://dx.doi.org/10.1016/j.compfluid.2005.07.005.

Youcef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

Youcef Saad and Martin H Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, July 1986. ISSN 0196-5204. doi: 10.1137/0907058.

A. H. Sameh and D. J. Kuck. On stable parallel linear system solvers. *J. ACM*, 25 (1):81–91, January 1978. ISSN 0004-5411. doi: 10.1145/322047.322054. URL http://doi.acm.org/10.1145/322047.322054.

Olaf Schenk and Klaus Gärtner. *PARDISO - User Guide Version 5.0.0*, 2014.

Olaf Schenk, Andreas Wächter, and Michael Hagemann. Matching-based preprocessing algorithms to the solution of saddle-point problems in large-scale nonconvex interior-point optimization. *Computational Optimization and Applications*, 36(2-3):321–341, 2007. ISSN 0926-6003. doi: 10.1007/s10589-006-9003-y. URL http://dx.doi.org/10.1007/s10589-006-9003-y.

Olaf Schenk, Matthias Bollhöfer, and Rudolf A. Römer. On large-scale diagonalization techniques for the anderson model of localization. *SIAM Rev.*, 50(1):91–112, February 2008. ISSN 0036-1445. doi: 10.1137/070707002. URL http://dx.doi.org/10.1137/070707002.

Gerald Schubert, Georg Hager, Holger Fehske, and Gerhard Wellein. Parallel Sparse Matrix-Vector Multiplication as a Test Case for Hybrid MPI+OpenMP Programming. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 1751–1758, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4577-6. doi: 10.1109/IPDPS.2011.332. URL http://dx.doi.org/10.1109/IPDPS.2011.332.

S. Shende and A. D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

S. W. Sloan. An algorithm for profile and wavefront reduction of sparse matrices. *Internacional Journal for Numerical Methods in Engineering*, 23:239–251, 1986.

Masha Sosonkina, Bora Uçar, and Yousef Saad. Hypergraph partitioning for parallel iterative solution of general sparse linear systems, 2007.

William F. Moss Stephen Demko and Philip W. Smith. Decay rates for inverses of band matrices. *Math. Comp.*, 43(43 (1984),):491–499, 1984.

Xian-He Sun. Application and accuracy of the parallel diagonal dominant algorithm. *Parallel Comput.*, 21(8):1241–1267, August 1995. ISSN 0167-8191. doi: 10.1016/0167-8191(95)00018-J. URL http://dx.doi.org/10.1016/0167-8191(95)00018-J.

Xian-He Sun, Hong Zhang, and Lionel M. Ni. Efficient tridiagonal solvers on multicomputers. *IEEE Trans. Comput.*, 41(3):286–296, March 1992. ISSN 0018-9340. doi: 10.1109/12.127441. URL http://dx.doi.org/10.1109/12.127441.

Fan Ye, Christophe Calvin, and SergeG. Petiton. A study of spmv implementation using mpi and openmp on intel many-core architecture. In Michel Daydé, Osni Marques, and Kengo Nakajima, editors, *High Performance Computing for Computational Science – VECPAR 2014*, volume 8969 of *Lecture Notes in Computer Science*, pages 43–56. Springer International Publishing, 2015. ISBN 978-3-319-17352-8. doi: 10.1007/978-3-319-17353-5_4. URL http://dx.doi.org/10.1007/978-3-319-17353-5_4.

A. N. Yzelman, Rob, and H. Bisseling. Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods. *SIAM Journal on Scientific Computing*, 2009.